NIC-TR-2004-016

NIC FastICA Implementation

Purpose

This document will describe the NIC FastICA implementation. The FastICA algorithm was initially created and implemented at The Helsinki University of Technology (HUT) by Hyvarinen and Oja [1]. The HUT version of the FastICA algorithm was implemented in Matlab, which is not fast enough for the problems required by the NIC and it cannot handle the problem sets required by the NIC. To overcome these issues, we have re-implemented the algorithm using the C and C++ programming languages, the LAPACK linear algebra package, the MPI message-passing library and the OpenMP library. The results of our re-implementation, including numerical validation using the HUT version of the FastICA code and the performance of the parallel implementations are included in this document.

For an explanation of the ICA process, see NIC-TR-2004-001.

The FastICA Algorithm

The FastICA algorithm attempts to find a set of independent components by estimating the maximum negentropy [2]. To make this estimate, the algorithm iteratively searches for the weight set $\mathbf{w}_i$ of a neural network from a data set $\mathbf{X}$ using the following procedure:

```
1 make a random guess for the weights associated with component $\mathbf{w}_{i,0}$
2 find $\mathbf{w}_{i,k} = E(\mathbf{x}g(\mathbf{w}_{i,k-1}^T\mathbf{x})) - E(g'(\mathbf{x}^T\mathbf{w}_{i,k-1}))\mathbf{w}_{i,,k-1}$, where x is an observation from
        $\mathbf{X}$ and g is a contrast function (see [2] for details).
3 normalize $\mathbf{w}_k$
4 if $|1 - |\mathbf{w}_k^T\mathbf{w}_{k-1}|| >$ tolerance repeat starting at step 2
5 repeat, starting at step 1, until all weights are found
```

The NIC implementation of the FastICA algorithm (see figure 1) whitens the source data as a preprocessing step then finds the weights associated with each independent component. The whitening process starts with the computation of the covariance matrix and the average for multichannel data. The covariance matrix is used to compute the sphering matrix (see NIC-TR-2004-001) and the data channel averages are used to center the data. The algorithm whitens the centered data by multiplying the data by sphering matrix.

After the data is whitened, the NIC FastICA code starts the search for the weight matrix by iteratively searching for each column of the weight matrix. The kth iteration of the search loop makes an estimate of the kth weight vector. The code can generate the estimate as a random vector, a user defined vector or the kth column of an identity matrix of the appropriate size, which is orthogonalized to the previously found weights and normalized.

The orthongonalized vector is then used as $\mathbf{w}_{k,0}$ in step 1 of the search algorithm, which is iteratively refined until $\mathbf{w}_{k,i}$ converges to an acceptable solution. The estimate is refined using the negentropy estimation, orthongonalized with the previous weights and normalized. After $\mathbf{w}_{k,1}$ is found, it is tested for converges by computing the inner product of $\mathbf{w}_{k,i}$ and $\mathbf{w}_{k,i+1}$. If the absolute value of the inner product equals 1 +/- ε, where ε is some tolerance, the estimate has converged and $\mathbf{w}_{k,i+1}$ becomes the kth column of the weight matrix.

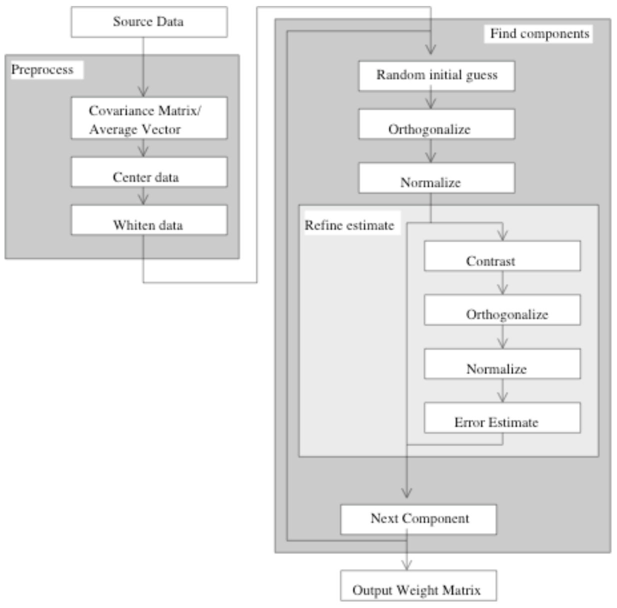

Figure 1. The FastICA algorithm whitens the source data in the preprocessing step, then iteratively finds the neural network weights associated with each component. On each iteration, an initial random guess of the weights for a component is made, then the guess is iteratively refined.

<u>Parallelizing FastICA</u>

All of the FastICA procedures (e.g., covariance matrix, center data, whiten data, etc), with the exception of orthogonalization and normalization, involve operations on the columns of the source data. As the order in which the columns are processed does not effect the results of the computation, we partition the source data into subsets and perform the computations independently. Once computations are complete on all partitions, we can collect the results to produce results for the whole data set. This characteristic of the FastICA algorithm makes the algorithm easily parallelizable for both shared memory and cluster computers. For example, we can partition the source data into subsets and distribute them to different processors. Each processor can compute the sum and sum squared of each data channel and the sum of the products of each data channel for its subset. We can accumulate the results from each processor to compute the covariance matrix for the entire data set. For a shared memory system, the partitioning involves unrolling the loop responsible for independent computations (i.e., the sum, sum squared and sum of products), from which we compute the covariance matrix. In the distributed algorithm, each processor performs the independent computations locally and a reduction operator produces the global sum, which the operator distributes to all nodes in the computation. Each node computes the covariance matrix independently. As the size of the data operations is large (~59000 observations on 32 channels), we expect both the shared memory and distribute systems to provide good speedup. If the number were too small, the distributed system would probably not do well. Figure 2 is a schematic description of the FastICA parallelization.

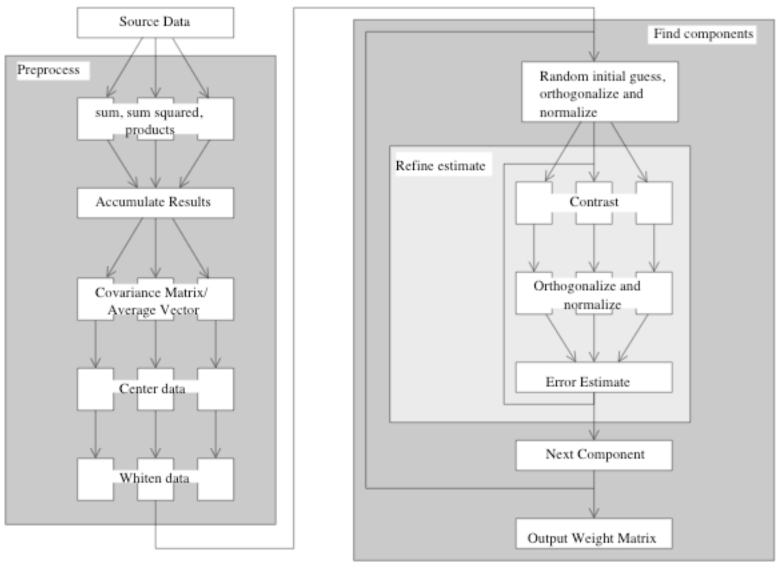

Figure 2 shows the structure of the FastICA MPI implementation. The openMP implementation has a similar structure.

Validation of FastICA

Sequential Validation

The results NIC FastICA implementation were compared to the results of the HUT FastICA implementation, which we assume is correct. To ensure consistency between the algorithms, we make a deterministic “initial estimate” of the weights by setting the guess for i^{th} weight to the i^{th} column of an n-by-n identity matrix, where n is the number of components to be found. The eigenvalue decomposition used by the HUT FastICA differs from the NIC FastICA, which leads to numerical inconsistencies (note: the differences are inconsistencies, not inaccuaracies). To avoid the numerical problems, we replaced the NIC FastICA’s sphering matrix with the sphering matrix produced by the HUT version. This ensures consistent whitening of the data.

To validate the code, we used the data set from experiment #5, (described in NIC-TR-2004-002), to generate a weight matrix using both implementations. The source data for this experiment consisted to ~59000 observations on 32 channels. As both implementations used the same sphering matrix, we assume that the whitening process in both cases produced the same results. We did not compare the output of these processes because of their size. Nevertheless, we can assume the NIC code is valid if the weight matrices produced by each version are the same within the numerical tolerances of the machine and software.

Parallel Validation

To validate the parallel versions of the FastICA algorithm, we followed the same testing procedures and validation criteria as in the sequential validation. We ran the procedures for 1 – 16 processors on the NIC's Dell Cluster (*neuronic*) {we will also run the code on 1 – 8 processors of the p655 shared memory cluster}. (A description of the clusters will be presented in a pending technical report).

The MPI FastICA implementation on one processor matched the HUT implementation exactly (within the precision of the machine). However, the results produced by *neuronic* show a difference of ~10^{-9} between the NIC and HUT version of FastICA when more than one processor was used. Moreover, the error grew as the number of processors was increased. This result is unexpected as there should have been no difference between the two implementations. A brief investigation into the architecture of the processors suggests that the hidden bits associated with the registers account for the discrepancy in accuracy. When MPI passes data between processors, these bits are truncated, hence a loss of accuracy. If the hidden bits are responsible for the discrepancy then it should not be of concern. FastICA is a fixed point algorithm, so the additional iterations should reclaim any lost accuracy.

Performance of Parallel NIC FastICA

Performance data for the parallel FastICA implementations were collected in conjunction with the validation tests. Timing instructions were added to the code for each implementation to measure the execution time for reading the data and for computing the weight matrix. The code was run using one processor per node (NodeT) and two processors per node (CPUT). The results we are presenting here are preliminary, however, they indicate that the distributed version of the FastICA algorithm performs well with some anomalous behavior (see Figure 2). Specifically, the dramatic increase in execution time in the NodeT case with 7 processors. The cause of the problems is under investigation. Also, the increase in execution time in the CPUT is surprising and under investigation.

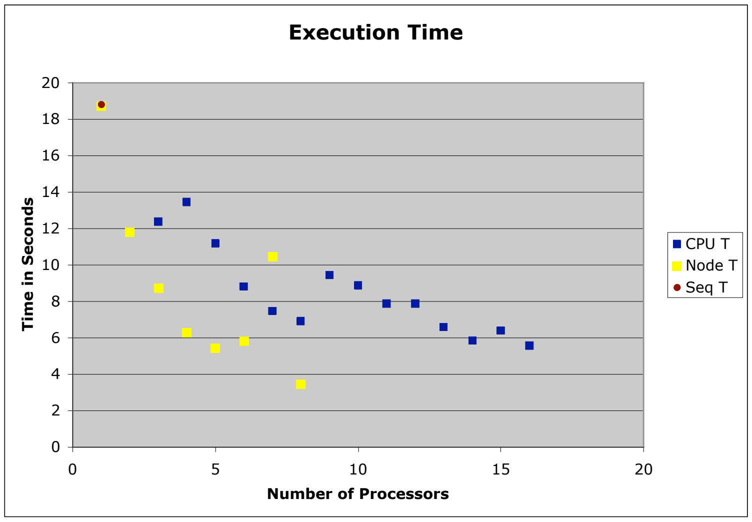

Figure 3 shows the performance evaluation on *neuronic* was run using 8 nodes. Two sets of performance data were collected. The NodeT set was collect using one processor per node and the CPUT set was collect using two processors per node.

The read time for the source data shows a steady, probably linear increase (see Figure 4). Though the performance does not seem to be severely impacted by the increase within the range of times the increase will limit code's scalability.

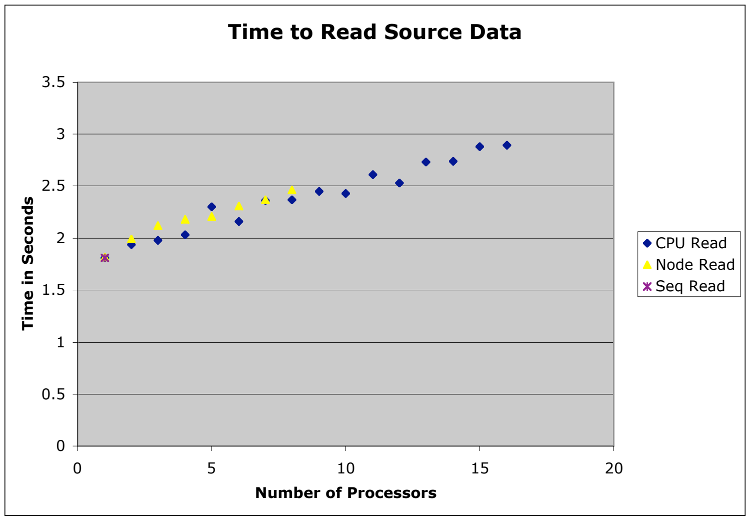

Figure 4 shows the time to read source data *neuronic* using up to 8 nodes. Two sets of performance data were collected. The NodeT set was collect using one processor per node and the CPUT set was collect using two processors per node.

Please note: this data is old and the algorithm was modified to address some efficiency issues. We will be altering this report as we get a more up-to-date set of performance data.

Bibliography

[1] Hyvärinen, A. and E. Oja. 1997, **A Fast Fixed-Point Algorithm for Independent Component Analysis.** *Neural Computation,* 9(7):1483-1492.

[2] Hyvarinen, A. 1999. **Fast and Robust Fixed-Point Algorithms for Independent Component Analysis**. *IEEE Transactions on Neural Networks* 10(3):626-634.