HiPerSAT Technical Report

Dan Keith, Christian Hoge, Robert Frank, and Allen D. Malony

University of Oregon
Neuroinformatics Center
Eugene, Oregon, USA
{dkeith, rmfrank, choge, malony}@cs.uoregon.edu

Abstract

*HiPerSAT, a C++ library and associated tools, processes large EEG data sets with statistical data whitening and ICA (Independent Component Analysis) methods. The library uses **BLAS**, **LAPACK**, **MPI** and **OpenMP** to achieve a high performance solution that exploits available parallel hardware. ICA is a class of methods for analyzing a large set of data samples and deducing the independent components responsible for the observed data. ICA is used in EEG signal analysis to understand neurological components of dynamic brain activity. We present two ICA implementations (FastICA and Infomax) that exploit parallelism to provide an EEG component decomposition solution of higher performance and data capacity than current MATLAB-based implementations. Experimental results and the methodology used to obtain them are presented. In addition, the integration of this functionality into the **MATLAB**-based **EEGLAB** tools [6] is described, as well as future plans for this research.*

1. Introduction

EEG (Electroencephalography)[1] is a technique for measuring changing electrical potentials on the scalp surface that occur as a result of dynamic brain function. Typically, the procedure involves placing multiple *sensors* on various regions of the scalp. These sensors have the ability to measure electric potential changes with microvolt sensitivity.

It is well-established [10] that electrochemical events within the brain can manifest as surface potential changes on the scalp. There are both clinical and research procedures which use this EEG data to infer physiological phenomenon, trauma and mental states. For example, EEG is used in the diagnosis and treatment of epilepsy, as well as to understand the cognitive basis of language.

Conventional EEG uses a small number (typically from 8 to 32) of sensors placed evenly about the scalp. Dense-array EEG ([21]) is a technique of using a fine-grained almost spherical mesh of sensors on the scalp, face and neck in order to provide an greater resolution of brain dynamics. Dense-array EEG uses anywhere from 64 to 256 sensors, each of which is sampled many times per second (250hz is a typical sampling rate). Collecting 15 minutes of EEG data from a 128-channel sensor mesh will result in more than 100Mb of data. Advances in EEG continue to increase both sensor density and sampling rate, resulting in even larger data sizes.

The challenge of using scalp-based data is that each sensor is actually measuring surface potential changes caused by a combination of underlying signals from various sources within the brain, as well as extra-brain sources. These signals are transmitted to the scalp via *volume conduction* through the various tissues and bone in the head. Thus, each sensor is actually receiving a mixture of different signals, and a given signal (e.g., from a firing neuron) may be received by several or all of the sensors, and at different intensities. In addition, there is a very low signal-to-noise ratio, with sources of noise including electrical equipment and physiological phenomenon such as eyeblinks, and heartbeats.

For meaningful information to be extracted from EEG data, the signal mixtures at each sensor must be separated into several components, each of which represents a more fundamental, independent physical cause or signal. Some of these components will cor-

[1] Most of the techniques described here for EEG apply equally well to the sibling technique of MEG (Magnetoencephalography), which measures magnetic rather than electrical fields.

respond to *artifacts* such as eyeblinks, heartbeats and in some cases, the experimental apparatus. A useful technique for separating these components from signal mixtures is Independent Component Analysis (ICA), where the extracted components describe temporally independent activities from spatially fixed overlapping sources.

By combining the high-resolution data provided by dense-array EEG with sophisticated data mining techniques such as ICA, researchers hope to develop ways to "see" into the neurophysiological and cognitive processes within the brain. However, the state-of-the-art in the EEG/MEG community is constrained by sequential execution of ICA algorithms within computational frameworks (e.g., MATLAB) with memory limitations and other overheads. Our work overcomes these limitations by providing high-performance, parallel implementations of two popular ICA algorithms, *FastICA* and *Infomax*. We have implemented a C++ library, *HiPerSAT* (High Performance Signal Analysis Toolkit), which provides functions that facilitate the separation of EEG data via both the FastICA and Infomax techniques. In addition, we have built a tool, `hipersat`, which allows easy access to this functionality from a command line. Finally, we have integrated HiPerSAT into the EEGLAB [7] framework, a MATLAB-based application that is used by neuroscientists to analyze and visualize EEG data.

The HiPerSAT implementation of these ICA methods provides a significant advantage over the existing MATLAB-based algorithms in two ways: time required for execution and maximum data size. The reduction in wall-clock time provided by HiPerSAT was one of the primary motivators for this research. However, an equally significant feature of HiPerSAT is its ability to handle much larger data sets than MATLAB can manage efficiently. The usual workaround for this problem in EEGLAB is to partition the dataset into smaller sets of samples. HiPerSAT is able to process large datasets efficiently without partitioning. Both of these space and time benefits of HiPerSAT become more significant as EEG hardware increases in channel density and sampling rate.

2. Background and Mathematical Foundations

Although the actual EEG measurements are captured via an electrical device that is mechanically and electrically attached to a biological organism's scalp, for the purpose of this paper we can abstract all of this into the realm of the digital. The primary function of an EEG device is to use analog-to-digital conversion hardware and software to discretize the continuous potential changes on the scalp over time into a fine-grained digital form. Modern dense-array EEG devices produce a stream of time-based data, typically sampled at 250hz. Each sample describes the voltage at each of the scalp sensors. In some cases, this voltage is expressed as an integer within a device-specific range. In other cases, the voltage is expressed as a real number indicating the actual voltage.

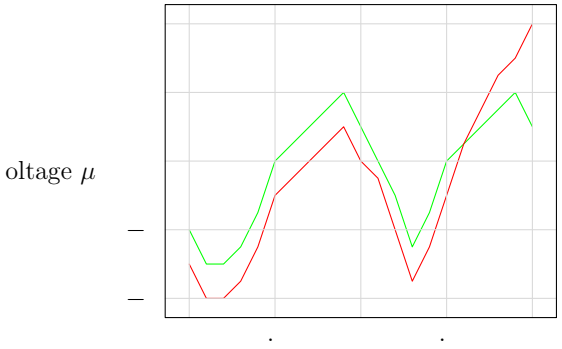


Figure 1. Sensor Voltage over time

Figure 1 illustrates how the voltage for a hypothetical 2-sensor EEG measurement varies over time. The horizontal axis corresponds to time, with each unit being a time sample (1/250 sec). The vertical axis is the voltage that was measured at that time. In this diagram, each channel is plotted against a common time and voltage coordinate system. Different curves (also known as traces) show the evolution of each channel's voltage over time.

Figure 2 is a graph showing the measured voltages for an actual 69-sensor EEG measurement sampled every 1/250second over a period of one second. Each channel is represented as a separate line or *trace* with its own y-axis, sharing a common time axis. This type of diagram is known as an *electroencephalogram* (EEG) and is the familiar two-dimensional representation of the multi-channel, time-ordered voltage data from the scalp sensors.

When this type of data is processed, it is digitized into a matrix form for ease of manipulation. We can formally describe the data output of an EEG device with n sensors as a time-ordered series of vectors $\mathbf{x}_t$, each of which has length n, where $x_t[j]$ is the voltage at sensor j at time t. This is best viewed as a rectangular data matrix, where each row (also known as a *channel*) represents a particular sensor's voltage over time, and each column corresponds to a particular time point. It is this data matrix that is the fundamental input for ICA methods. This sensor data matrix $\mathbf{x}$ contains all of the measured scalp data over time.

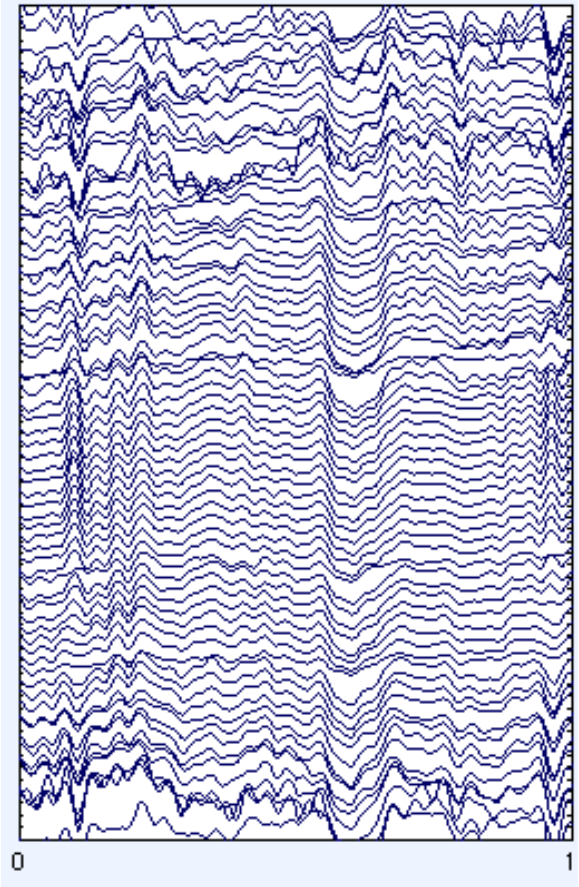


Figure 2. One second of 69-channel EEG

In component analysis, the potentials measured in $\mathbf{x}$ are assumed to be mixtures of one or more underlying fundamental signal components. The proportion of each component's activity measured at a sensor is determined by the component's *scalp map*, which is the vector function which describes how a given component's influence is spread out between the various sensors. Different component analysis methods make different assumptions about the relationship between these signal mixtures and their sources.

The goal of ICA as applied to EEG data is to explain the observed matrix $\mathbf{x}$ of signal mixtures in terms of two other quantities:

- $\mathbf{s}$ - A matrix of time-ordered values corresponding to posited independent *source signal* components
- $\mathbf{A}$ - A *mixing* matrix that accounts for how the independent signal components in $\mathbf{s}$ are mixed into the observed scalp *signal mixtures* $\mathbf{x}$.

The columns of $\mathbf{A}$ describe spatial distribution of component activity.

Formally, we wish to solve the Equation 1 for both $\mathbf{A}$ and $\mathbf{s}$, given that we have a measured set of mixtures $\mathbf{x}$:

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{1}$$

In order to derive $\mathbf{A}$, it is sufficient to derive $\mathbf{W}$, also known as the *unmixing matrix*, such that:

$$\mathbf{A} = \mathbf{W}^{-1} \tag{2}$$

$$\mathbf{s} = \mathbf{W}\mathbf{x} \tag{3}$$

Most ICA algorithms are based upon finding the *unmixing* matrix $\mathbf{W}$, as it is this matrix that allows the independent components $\mathbf{s}$ to be extracted from the set of signal mixtures $\mathbf{x}$.

Without any other constraints or assumptions, Equation 1 and Equation 2 have an infinite number of solutions. However, ICA makes a few critical assumptions that restrict the set of solutions to a small set of possible solutions, ideally with a single *most-likely* solution. ICA assumes that the input data are a mixture of temporally independent components whose sources are spatially fixed over time. This means that knowledge of $\mathbf{s}_t[i]$ for a given sample $\mathbf{s}_t$ provides no information about $\mathbf{s}_t[j]$.

ICA relies upon the fact that the *probability distribution function* (*pdf*) of a truly independent component is non-gaussian, whereas the pdf of a mixture of components follows a gaussian distribution. This result derives from the *Central Limit Theorem* of statistics, which says that the sum of a set of independent random variables has a gaussian pdf. Because a signal mixture is actually a sum of its independent components, we would expect that a mixture is more gaussian than a component that contributes to it. These constraints are used by ICA methods to determine $\mathbf{A}$ and $\mathbf{s}$, given a sufficiently large set of samples $\mathbf{x}$. Most ICA methods require that the number of samples exceeds several times the square of the number of channels. Detailed information on the mathematics underlying ICA is available in [1] and [19].

An example of how the independent components are extracted from signal mixtures is shown in Figure 3. The left figure shows a signal mixture of four independent components (in this case, distinct synthetic sinusoidal signals). The right figure shows the resultant independent components as extracted by FastICA. What is remarkable is how the underlying components are discovered within the apparent chaos of the input mixtures.

2.1. Data Whitening

Both of the ICA algorithms that we have chosen to implement, FastICA and Infomax, make the assump-

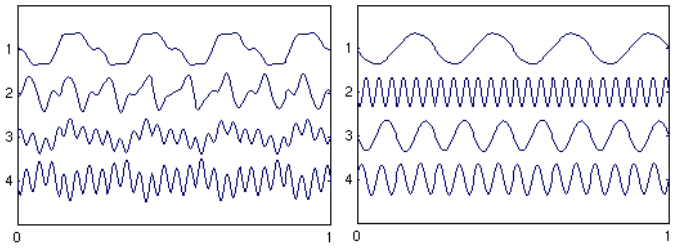


Figure 3. EEG Mixture and Components

tion that the input data $\mathbf{x}$ has been *whitened*. *Whitening* or *sphering* data is a process whereby the original mixture data $\mathbf{x}$ is multiplied by a matrix $\mathbf{S}_{ph}$ (the sphering matrix) to produce a set of *whitened* data that is uncorrelated with 0 mean and unity variance.

In Equation 2 above, the unmixing matrix $\mathbf{W}$ is applied to the mixture data $\mathbf{x}$ to generate the independent components. Because the ICA algorithms assume that $\mathbf{x}$ has been whitened, it is necessary to preprocess the data $\mathbf{x}$ by applying the sphering matrix $\mathbf{S}_{ph}$ to $\mathbf{x}$. This produces a set of whitened data $\mathbf{x}' = \mathbf{S}_{ph}\mathbf{x}$ which is amenable to the ICA algorithms (Equation 4).

$$\begin{aligned} \mathbf{s} &= \mathbf{W}\mathbf{x} && (4) \\ &= \mathbf{W}\mathbf{S}_{\mathbf{ph}}^{-\mathbf{1}}\mathbf{S}_{\mathbf{ph}}\mathbf{x} && (5) \\ &= \mathbf{W}_{\mathbf{gt}}\mathbf{x}' && (6) \end{aligned}$$

where $\mathbf{W}_{\mathbf{gt}}$ is known as the *weight matrix* and $\mathbf{x}'$ is the whitened data. Because $\mathbf{W}_{\mathbf{gt}}$ is an orthogonal matrix, it acts as an additional constraint on the constrained optimization problem that is the heart of ICA, thus speeding convergence to a solution.

2.2. Infomax

One of the earliest ICA algorithms was described by Bell and Sejnowski [3] as an *information-maximization* or *infomax* algorithm. Infomax derives a weight matrix $\mathbf{W}_{\mathbf{gt}}$ that maximizes the statistical independence of the components by using an algorithm which minimizes the redundancy amongst outputs of a neural net. This is done by using a sigmoidal nonlinearity to estimate the data's higher-order moments. This ensures that the resultant components are maximally independent and non-gaussian.

2.3. FastICA

Infomax is the primary algorithm used within EEGLAB. Another form of ICA is encapsulated in the FastICA method, first described in [16] and implemented in MATLAB as [14]. FastICA works by searching for a weight matrix that maximizes the non-gaussianity of the resultant components. Any mixture of non-gaussian random variables will be more gaussian than the variables themselves (Central Limit Theorem of statistics). Therefore, it is possible to use non-gaussianity as a measure of statistical independence. FastICA uses this fact by building up a weight matrix column-by-column, where each column maximizes the non-gaussianity of the corresponding component.

Non-gaussianity and therefore, independence, is maximized indirectly by computing and maximizing a *contrast function*. Different contrast functions can be used, although the use of *kurtosis* and *negentropy* have been shown to provide a good trade-off of speed and reliable convergence.

3. Related Work

Tucker et al [21] at the University of Oregon have pioneered and advocated the use of dense-array EEG measurements. Makeig's group at SCCN (*Swartz Center for Computational Neuroscience*) [10] has led the field in the use of ICA for discovering underlying components and for removing artifacts from such data. This group has developed the EEGLAB toolkit [7] that offers EEG analysis and visualization, including data analysis based on various ICA algorithms. In particular, the `runica()` function within EEGLAB is an improved implementation of the *infomax* algorithm as described by Bell and Sejnowski [3]. It is this MATLAB version of the algorithm that was used as a standard of correctness when implementing HiPerSAT's parallel, C++ version.

ICA and the techniques known as *factor analysis* and *principal component analysis* (PCA) are both forms of *blind source separation*, which is the problem of determining the components sources in a set of mixtures, without having prior knowledge of how these sources are mixed. These algorithms solve essentially the same problem, but make different assumptions about the underlying data. ICA is different from PCA in that it minimizes the correlation of higher-order statistical moments. The blind source separation problem is also known as the *Cocktail Party Problem* [13] because of the way that a human brain can pick out the distinct conversations occurring simultaneously at a hypothetical cocktail party with multiple concurrent conversations.

Information about ICA methods can be found in [1] and [19], and several of the sources mentioned in the bibliography.

4. Architecture of HiPerSAT

The immediate goal of HiPerSAT is a high-performance implementation of two distinct ICA algorithms, FastICA and Infomax. Longer term, we envision HiPerSAT as a library framework for a family of EEG analysis methods, including wavelet analysis. The HiPerSAT design is based on a common set of utility code for I/O, data structures, and mathematical and statistical computations. In addition, it supports flexible parallelism models that can be reused depending on the analysis algorithm requirements.

The HiPerSAT library provides processing modules corresponding to the data transformations of Whitening, FastICA and Infomax, respectively. A given invocation of HiPerSAT will combine the Whitening module with either FastICA or Infomax, which will operate upon the whitened data. The HiPerSAT library and tools are C++ code that builds upon several different third-party technologies. These third-party technologies can be broken into two categories:

1. Mathematical libraries and data structures - BLAS, LAPACK, ATLAS
2. Interprocess communication and parallelism support - MPI, OpenMP

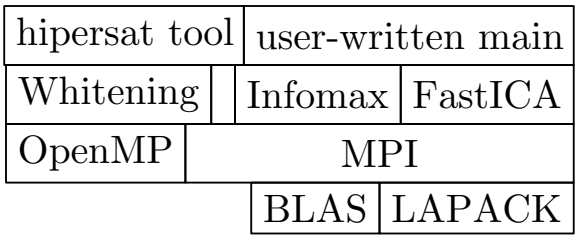


Figure 4. Component-level architecture of HiPerSAT

The HiPerSAT component-level architecture is illustrated in Figure 4. Basically, there are three independent signal processing modules corresponding to the data transformations of Whitening, FastICA and Infomax, respectively. Typically, a given invocation of HiPerSAT will combine the Whitening module with either FastICA or Infomax, which will operate upon the whitened data.

4.1. Parallelism Mechanisms

HiPerSAT is capable of solving large problems using FastICA or Infomax running in sequential mode without requiring any parallel capabilities. However, the implementations achieve greater performance (see Section 9 below) when they are run in a parallel-capable environment. The current HiPerSAT implemention of FastICA relies upon MPI (running multiple processes) for its parallelism, whereas HiPerSAT's Infomax implementation uses OpenMP (single-process, shared memory, multiple processors). Our initial choice of parallelism method is due to dependency analysis of the original algorithms.

There are several implementations of MPI available, including Intel's MPI, and the open-source MPICH [11] and LAM-MPI [12] [18]. The HiPerSAT performance results for FastICA we report here are based upon MPICH. OpenMP [17] is compiler-dependent and we make use of different OpenMP-compatible compilers in our work, including IBM's xlC and Intel's icc.

4.2. Mathematics Libraries

Much of the matrix, vector, and linear algebra functionality needed by HiPerSAT is provided by two well-known linear algebra libraries, BLAS (Basic Linear Algebra Subprograms) [9] and LAPACK (Linear Algebra PACKage) [2]. Modern platforms provide high-performance, C-accessible, threadsafe implementations of both LAPACK and BLAS. The HiPerSAT implementation uses the Intel MKL (Math Kernel Library) or IBM ESSL (Engineering and Scientific Software Library) to provide high-performance versions of BLAS and LAPACK.

5. Overview of HiPerSAT Usage

The basic tasks of HiPerSAT are:

1. Obtain input data, in the form of a matrix $\mathbf{x}$.
2. Whiten data by computing sphering matrix $\mathbf{S}_{ph}$ such that $\mathbf{S}_{ph}\mathbf{x} = \mathbf{x}'$, where the rows in $\mathbf{x}'$ are uncorrelated with each other.
3. Search for a weight matrix $\mathbf{W}$ such that will unmix the whitened signal mixtures $\mathbf{x}'$ into a set of independent source signals $\mathbf{s}$.
4. output the computed weight matrix, sphering matrix, mixing matrix $\mathbf{A}$, and independent components obtained by applying the weight matrix W_{gt} to the whitened signal mixtures $\mathbf{x}'$.

The HiPerSAT library is a set of C++ classes used to perform these steps. The `hipersat` program uses these classes in a utility accessible via a command line.

5.1. Command Line Utility

The behavior of the `hipersat` program is specified via a parameter file which is passed as an argument to `hipersat`. For example, the command:

```
> hipersat myICA.txt
```

will cause the `hipersat` program to read ICA parameters from file "myICA.txt". These parameters include such information as the names of the input and output files as well as various parameters such as algorithm (FastICA vs Infomax), learning rate and convergence tolerance.

The `hipersat` program will initiate the sequential or parallel execution and then invoke the Whitening, FastICA, and Infomax methods in the correct order, based upon the values in the the parameter file.

5.2. Integration into EEGLab

EEGLAB is a widely-used neuroscience application produced by SCCN [7] that provides visualization, filtering, and analysis of EEG data in a powerful, GUI-based environment. One of the features of EEGLAB is the ability to perform one of several ICA methods upon EEG data. By default, EEGLAB will execute a MATLAB-based version of Infomax.

We have extended EEGLAB's ICA calling interface to enable the convenient execution of HiPerSAT's Infomax and FastICA versions. Normally, an EEGLAB user loads an EEG data file and executes EEGLAB's `runica` method. If the user instead selects either `nic-fastica` or `nic-infomax` as the type of algorithm, then the EEG data will be exported to a disk file and the `hipersat` tool will be invoked. After `hipersat` completes, the resulting independent components and other output data are imported back into EEGLAB for subsequent display and analysis.

6. FastICA Execution

As mentioned above, HiPerSAT implements two different algorithms, FastICA and Infomax, each using their own parallelism mechanism, MPI and OpenMP, respectively. This section describes the FastICA implementation in detail. Section 7 describes the details of the Infomax implementation. The overall dataflow in FastICA is:

1. start processes on head and remote nodes
2. load parameters and replicate to workers
3. load data and distribute amongst workers
4. optionally whiten the data
5. compute weights using FastICA algorithm
6. output components, weights, sphering matrix

6.1. Process Creation

The HiPerSAT FastICA/MPI implementation assumes that there are one or more separate processes that run and communicate via MPI. The MPI execution model that we use in FastICA assumes that there is one machine or node (the *head node*) that is the machine where the user will initiate execution of a FastICA run of HiPerSAT. The user will initiate execution by using the MPI-provided `mpiexec` command, which ensures that the `hipersat` command will be executed on the head node and on a set of specified additional *worker nodes*, separate machines where HiPerSAT is installed. The `mpiexec` command uses various mechanisms (e.g., `ssh`, `rsh`, `exec`) to securely log in as the initiating user on the worker nodes and start up replicas of `hipersat` on each node. It is possible to run several instances of `hipersat` on a given node, although CPU-level parallelism is reduced in this case. After `mpiexec` has created processes on the head node and any worker nodes, the execution of the `hipersat` main program will begin in each of these processes.

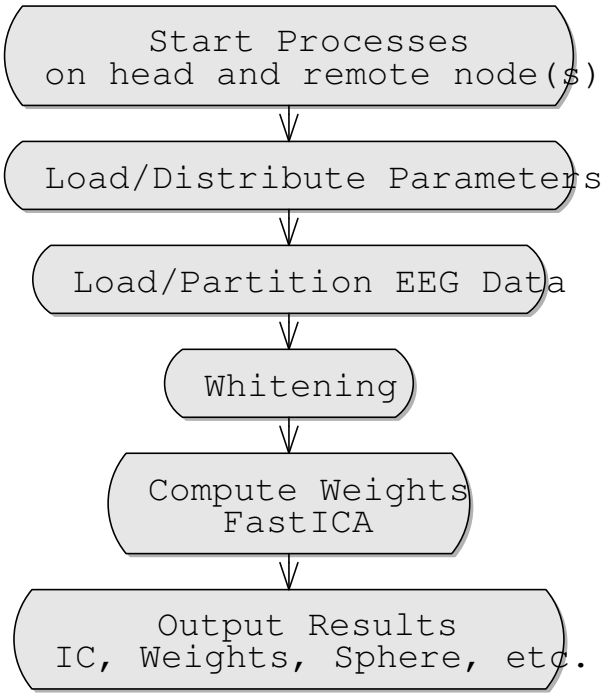


Figure 5. Dataflow in FastICA/MPI

The FastICA/MPI algorithm outlined above requires that the head node is distinguished during import and export of the EEG data and parameters, but during the actual computation of weights there is virtually no distinction made between the initiating head node and the worker nodes. Effectively, the head node *is* a worker node. In the description of the algorithm below, most references to worker nodes will refer to either the head node or a worker node. Where the head node and worker nodes have different behavior, this will be noted.

6.2. Parameter Input and Distribution

Because only the head node is able to read the input parameter file (no shared filesystem is assumed), it is necessary for the relevant parameters to be distributed to all of the worker MPI processes so that the head node and workers have a common set of operating parameters. This step is performed before the actual data file(s) are loaded and processed.

Early in the execution of HiPerSAT, the head node uses `MPI_BCast` to transmit a copy of the parameter file to the workers. After this point, the head nodes and workers have a common set of operating parameters.

6.3. Data Input and Distribution

Similar to the parameter distribution mechanism above, the actual input EEG data file must be distributed to the workers by the head node (no shared filesystem is assumed).

In the FastICA algorithm, if we have m samples in the input data set, and there are p total MPI processes, then each worker will get m/p samples from the original data file. *worker*0 (the head node) will have samples 0 through $m/p - 1$, *worker*1 will have sample m/p through $2m/p-1$, and so on. If the number of samples is not evenly divisible by the number of MPI processes p, then the remainder of the samples will be given to a subset of the processes (i.e., some processes will have $m/p + 1$ samples). The subset of samples allocated to a worker is called a *partition*.

The purpose of this phase of FastICA processing is to ensure that the entire input file is loaded into an effectively *distributed memory*, which in this case spans multiple processes. No single process has the entire EEG data in its address space; instead, the set is partitioned amongst the head node and workers. Many of the subsequent stages in the algorithm are performed in parallel, with each `hipersat` process executing independently with periodic data exchange and synchronization between the processes.

6.4. Data Whitening

The whitening of the input data is parallelized in the HiPerSAT FastICA algorithm. Here, each worker computes the covariance of its partition in parallel, and then combines the resultant matrix with a call to `MPI_Reduce`, which sums the various per-worker covariance matrices.

The time for data whitening is neglible compared to the time required for either FastICA or Infomax. Hence, we do not report on the whitening time in this document.

6.5. FastICA

After the data has been loaded and whitened (either within HiPerSAT or prior to execution), each worker process will contain a whitened version of its partition's data. At this point, the FastICA algorithm properly begins with each worker process executing the method `searchForWeights()`, which is responsible for computing a weight matrix $\mathbf{W}$ that properly separates the whitened signal mixtures into independent components.

The method `searchForWeights` described in Algorithm 1 will compute $\mathbf{W_{gt}}$ one row at a time, with each row $\mathbf{w}$ corresponding to the weight vector that extracts one particular component from $\mathbf{x}'$. Let n be the number of input signal mixtures and the number of desired independent components; $\mathbf{W_{gt}}$ is therefore a $n \times n$ matrix and $\mathbf{w}$ is a vector of length n. Let m be the number of actual data samples in $\mathbf{x}'$.

Algorithm 1 FastICA searchForWeights

Require: $\mathbf{x}'$ has been whitened
Ensure: $\mathbf{s} = \mathbf{W_{gt}}\mathbf{x}'$ and $\mathbf{s}$ is maximally independent.
  {$\mathbf{P}$ is projection matrix} {$\mathbf{W_{gt}}$ is output weight matrix}
  $\mathbf{P} \Leftarrow$ makeZeroMatrix(n)
  $\mathbf{W_{gt}} \Leftarrow$ makeZeroMatrix(n)
  {For each desired component...}
  for $i = 0$ to $n - 1$ **do**
    MPI Broadcast of $\mathbf{w}$
    $\mathbf{w} \Leftarrow$ computeWeight($\mathbf{P}$, i, $\mathbf{x}'$)
    $\mathbf{P} \Leftarrow$ Rank1Update($\mathbf{P}$, $\mathbf{w}$)
    $\mathbf{W_{gt}} \Leftarrow$ addWeightToMatrix($\mathbf{P}$, $\mathbf{w}$)
  end for

Algorithm 1 will initialize two square matrices $\mathbf{P}$ and $\mathbf{W_{gt}}$ and will then compute the n weights necessary to fill in $\mathbf{W_{gt}}$. The projection matrix $\mathbf{P}$ is maintained to ensure that `computeWeight()` can generate a new

weight that is orthogonal to all of the previously found weights.

The `Rank1Update` function updates the projection matrix $\mathbf{P}$ with the computed version of $\mathbf{w}$. The `addWeightToMatrix` function simply copies the weight vector into the resultant weight matrix.

From a computation point of view this function is interesting in that it shows that the `computeWeight` function is called n times. Except for the trivial linearity in n, the function `computeWeight` described in Algorithm 2) will contain all of the computational complexity.

Algorithm 2 FastICA computeWeight(**B**, i, **x**)

```
w ⇐ initializeWeight( i )
w ⇐ orthogonalize( w, B )
w ⇐ normalize( w )
{w is now a starting weight vector}
{Use Fixed-point method (ala Newton) to converge this component's weight}
delta ⇐ 100000
while delta ≥ tolerance do
  w' ⇐ improveWeight( w, x )
  MPI Reduce - Sum, then share, all the w
  delta ⇐ estimateConvergence( w, w' )
end while
```

Algorithm 2 (`computeWeight`) will perform a number of iterative calls to `improveWeight`, which takes an existing weight vector and the entire data matrix and computes a better weight vector. This iterative process (based upon Newton's method) of weight vector improvement continues until a user-specifed number of iterations has passed, or a fixed-point has been met within some user-specified tolerance. The `estimateConvergence` function is trivial and is $O(n)$; it simply computes a distance function between $\mathbf{w}'$ and $\mathbf{w}$.

This means that the bulk of the performance cost is in the `improveWeight` algorithm, described in Algorithm 3. The mathematical basis for this algorithm is derived in [16] [15]. Essentially, there is a fixed point convergence relation (Equations 7,8) that uses a gradient method to maximize the non-gaussianity of the product $\mathbf{w}'\mathbf{x}$, which is the projected independent component extracted by $\mathbf{w}'$.

$$\mathbf{w}' = E\{\mathbf{x}g(\mathbf{w}^\mathbf{T}\mathbf{x})\} - E\{g'(\mathbf{w}^\mathbf{T}\mathbf{x})\}\mathbf{w} \quad (7)$$

$$\mathbf{w}' = \mathbf{w}'/\|\mathbf{w}'\| \quad (8)$$

Because the expectation of these random variables is unknown *a priori*, the algorithm relies upon sampling the available data to get an estimate of the above expectations. This will require traversal of the data samples, and is one of the reasons why the algorithm takes longer as more samples are added. Because of the properties of whitened data and non-gaussian components, the generalized Equation 7 can be simplified into the actual formula that is used to compute new weights within the FastICA algorithm. This eliminates several matrix-matrix multiplies and matrix inversions, reducing some of these to rank-one updates (a much simpler matrix operation). This derivation is detailed in the various FastICA references [14].

The function $g()$ is known as the *contrast* function and it and its derivative $g'()$ are used to evaluate the *gaussianity* of a signal component as generated by $\mathbf{w}'\mathbf{x}$. HiPerSAT supports three different contrast functions: *cubic*, *tanh*, and *gaussian*. The *cubic* contrast function is used to measure the *kurtosis* of the separated components, the *tanh* contrast function measures the *negentropy* of the components, and the *gaussian* function measures the *gaussianity* of the components.

The HiPerSAT implementation contains a version of `improveWeight` for each of these contrast functions. Each version of `improveWeight` shares the same basic loop structure, with subtle differences in how the contrast function and its derivative are used for each element in the loop. From a computation point of view, however, we only need concern ourselves with the scaling considerations due to the loop structure.

Algorithm 3 FastICA improveWeight(**w**, **x**)

```
Ensure: Return improved weight w'.
w' ⇐ w
{For each sample 1 ... m}
for i = 0 to m − 1 do
  {For each component 1 ... n}
  for j = 0 to n − 1 do
    w'[j] ⇐w'[ j ] + contrast( w, x, j )
  end for
end for
```

The actual `contrast()` function used in Algorithm 3 will be based upon one of the three supported versions of $g()$ and g' in Equation 7 (cubic, tanh, and gaussian).

6.6. Result Gathering and Output

After FastICA successfully converges on a weight matrix that satisfies the convergence criteria specified by the problem, HiPerSAT will optionally export to disk the discovered weight, sphering, mixing and unmixing matrices.

HiPerSAT can optionally export the separated components as an output EEG file. Because each worker has a separate partition of the input data, exporting the separated components is performed by the head node by gathering each worker's partition of separated component data and writing it to disk.

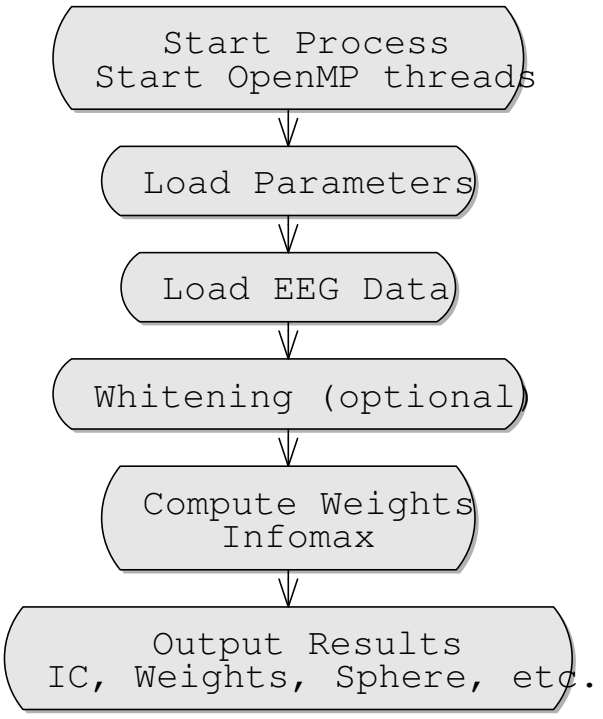


Figure 6. Dataflow in Infomax/OpenMP

7. Infomax Execution

The Infomax implementation within HiPerSAT relies upon OpenMP for its parallelism. One of the potential advantages of an OpenMP approach is the ability for each separate thread to share the input EEG data, as well as access any shared data structures (e.g., the current weight matrix).

The execution of Infomax is outlined below:

1. start process
2. load parameters
3. load data
4. optionally whiten the data
5. create one or more worker threads
6. compute weights using Infomax algorithm
7. output components, weights, sphering matrix

7.1. Parameter and Data Input

The parallel workers of HiPerSAT/Infomax are implemented as threads within a single process, with each thread having access to the same shared data in memory. This obviates the need to explicitly partition and distribute the parameters and data amongst the workers; similarly, the exporting of the separated components does not require that the partitions are gathered prior to exporting. This makes reading in the parameters and input data a trivial operation. In the Infomax/OpenMP algorithm, a single process contains the entire EEG input data set.

7.2. Data Whitening

Data whitening in Infomax is identical to the standalone (non-parallel) version of data whitening used in FastICA. The whitening process is not yet parallelized in Infomax, but may be in future versions.

7.3. Infomax

After the input has been loaded and whitened, the HiPerSAT process will contain the entire input data set in memory, accessible to all of the worker threads. At this point, the Infomax algorithm begins with `searchForWeights()` as described in Algorithm 4. After initializing the per-process data structures, the algorithm splits into several concurrent threads, each of which has a distinct value for the `pid` variable.

Each of these threads will perform identical work until parallelized loops are executed within `TrainNN` (see Algorithm 5). It is this `TrainNN` function which is measured when the Infomax iteration cost is reported in the graphs and tables in this document.

Algorithm 5 describes `TrainNN` in more detail. This function works by partitioning the input data into disjoint rectangular blocks consisting of the n channels for a small (approximately 35) number of samples. The ordering of these blocks is randomized as part of the algorithm to ensure that any temporal bias in the data is eliminated.

As each parallel thread proceeds, it will obtain the same block as the other threads (no parallelism yet). At this point, however, the chosen block (called $\mathbf{x}$) is traversed in parallel such that each OpenMP thread will operate on a fraction of the columns within a block. For example, if the block $\mathbf{x}$ has 36 columns and there are 4 threads, then thread 1 might get columns $1, 5, 9, \ldots$, thread 2 might get columns $2, 6, 10, \ldots$ and so on. [2]

[2]This ability to parallelize a loop is one of the advantages of

Algorithm 4 Infomax searchForWeights()

Require: $initialW$ = random matrix of weights
Require: $tolerance$ = user-specified tolerance
Require: x = matrix of input signal mixtures
  W $\Leftarrow$ **initialWMatrix**
  oldW $\Leftarrow$ **W**
  WDelta $\Leftarrow makeZeroVector(n)$
  oldWDelta $\Leftarrow makeZeroVector(n)$
  bias $\Leftarrow makeZeroVector(n)$
  initDelta $\Leftarrow makeZeroVector(n)$
  OpenMP - In Parallel, for each thread pid
  while $delta > tolerance$ **do**
    trainNN(pid)
    if $pid = masterPid$ **then**
      OpenMP - Master only, other threads wait
      adjustLearning(pid)
    end if
  end while

Within this first parallel `for` loop, we are computing a new rectangular matrix **u** which is the same size as **x**. This **u** matrix will contain the projected components **Wx**, plus a bias vector **b**. The matrix multiplications within this parallel `for` loop will occur in parallel, affording a p-way speedup for this section of code if there are p processors available for OpenMP threads.

The next loop is also a parallel `for` loop which applies an *activation function* to the biased component block **u**, resulting in a new block matrix such that **y** $\Leftarrow$ `activation( u )`. This activation function is used to evaluate the biased components in **u** to determine how the current weight matrix candidate **w** should be modified.

Finally, the `updateWeights` function will use the bias vector **b** to compute a new version of **w** and will update the weight matrix **W** accordingly. This contains a minor parallel loop and then a master-only loop.

7.4. Result Output

Writing out the separated components is currently implemented in the HiPerSAT version of Infomax in a straightforward, yet memory-wasteful, fashion. Currently, a new matrix as big as the input data matrix is allocated and filled with the separated components. Then, this matrix is exported to disk. An obvious improvement planned for HiPerSAT Infomax is to compute the separated components on the fly as they are output. This would reduce the memory requirements

the OpenMP approach. It relies upon each of the per-column operations being independent of every other one.

Algorithm 5 Infomax trainNN(pid, X)

  for $outer = 1 \ldots numBlocks$ **do**
    for $blockCol = 1 \ldots blockCols$ **do**
      OpenMP - Parallel for loop body
      $index \Leftarrow shuffle(outer + blockCol)$
      x $\Leftarrow$ **X**[$index$]
      w $\Leftarrow$ **W**[$index$]
      {u = W*x + bias}
      u[$blockCol$] $\Leftarrow$ **Wx**
      u[$blockCol$] $\Leftarrow$ **u**[$blockCol$] + **bias**
    end for
    $numBlockCells \Leftarrow blockRows \times blockCols$
    for $blockCell = 1 \ldots numBlockCells$ **do**
      OpenMP - Parallel for loop body
      y[$blockCell$] $\Leftarrow$ activation(**u**[$blockCell$])
    end for
    updateWeights(**w**, **y**, **biasLoad**)
  end for

Algorithm 6 Infomax updateWeights(**w**, **y**, **bl**)

  for $blockCell = 1 \ldots numBlockCells$ **do**
    OpenMP - Parallel for loop body
    bl[$blockCell$] $\Leftarrow$ 1 - 2**y**[$blockCell$])
  end for
  if $pid = masterPid$ **then**
    OpenMP - Master only, other threads wait
    tmp $\Leftarrow zeros(n)$
    for $i = 0 \ldots blockCols$ **do**
      for $j = 0 \ldots n$ **do**
        $tmp[j] \Leftarrow tmp[j] + bl[i, j]$
      end for
    end for
    return $max(\mathbf{w}) > MAXWEIGHT$)
  end if

during the output phase by 50% and would likely improve performance during this phase.

8. Experimental Methodology

We verified the validity and efficiency implementations of these algorithms with a series of tests. The validity tests ensured that we gave the correct answers, as defined by the EEGLAB implementations of these algorithms. We have validated that the results are virtually identical for a variety of both real and synthetic data sets. We have compared the results of the EEGLAB versions of Infomax and FastICA with the corresponding HiPerSAT results and they match almost exactly.

The efficency tests evaluate the performance of the algorithms on various hardware/software configurations, as well as to understand how this performance changes based upon the dimensionality of the data and the processing parameters. We focus on execution time efficiency. We performed timed tests on a variety of hardware/software configurations, upon a variety of data set sizes. The platforms that we tested are listed in Table 1.

Table 1. Tested HiPerSAT Platforms

Name	Processor(s)	Parallelism
Mac G4	PowerPC G4	sequential only
Neuronic	Xeon (2xCPU)	16xMPI/4xOpenMP
P655 (IBM)	8x1.5Ghz POWER4	8xOpenMP/8xMPI
P690 (IBM)	16x1.3Ghz POWER4	16xOpenMP/16xMPI
Optix (SGI)	16x1.5Ghz Itanium2	16xOpenMP/16xMPI

The primary data set used was from a psychology study in which subjects were given various stimuli and asked to perform a reading task. The purpose of the experiment is to determine whether there are particular electrical sources within the brain during certain evoked neural states, and if so, to isolate these components from the mixture data available at the scalp via EEG.

The basic metric of the input data size is its dimensionality, in terms of number of channels and number of samples. The expectation is that the problem complexity is proportional to the square of the number of channels and linearly proportional to the number of samples. The number of channels used in these experiments varied from 64 to 256, whereas the number of samples ranged from $100,000$ to $10,000,000$. As a point of reference, a complete ICA decomposition using the standard MATLAB implementations ranges in wall clock time from 1 hour to several days depending upon the dimensionality of the data set and the execution platform.

The two ICA algorithms (FastICA and Infomax) are both based upon an iterative method that achieves convergence. The number of iterations varies depending upon the content of the data its fixed dimensionality. Therefore, different data sets of the same size could take wildly different amounts of time to converge to a solution. This makes the use of overall wall clock time a less useful metric for comparing performance between different algorithms and data sets.

In order to address the above deficiency, we use a per-iteration time cost metric that is invariant between different data of a given dimensionality. This per-iteration cost, C_{iter}, is used in the experimental results described here. This permits more experiments to be run, because the program is not run until convergence, allowing us to explore a broader evaluation space.

For FastICA, the iteration time measured corresponds to `searchForWeights()` described in Algorithm 1. The iteration time measured for Infomax corresponds to the time spent in the `searchForWeights()` function described in the abstract parallel implementation described in Algorithm 4 above. In both cases, C_{iter} is computed as $C_{iter} = totalTime_{iter}/numberOfIters$, where $totalTime_{iter}$ is the measured time spent in `searchForWeights`.

In both FastICA and Infomax, we used the parallel performance analysis software TAU (Tuning and Analysis Utility) [20]. The TAU software allowed us to instrument the HiPerSAT source code to facilitate the gathering of fine-grained performance information for both MPI and OpenMP.

9. Experimental Results

Testing HiPerSAT proved to be challenging because of the potential for an overwhelming set of combinations of data file size, hardware (see Table 1), algorithm (Infomax vs. FastICA), parallel processing type (OpenMP vs. MPI vs. standalone), and processing parameters (learning rate, convergence threshold, etc). For the purposes of this paper, we will focus on those results most salient to the issue of performance gains possible via parallelism. We have restricted our analysis here to a standard number of samples ($111,000$ samples) in order to eliminate variability due to the number of samples. Parallelism performance improves with larger numbers of samples.

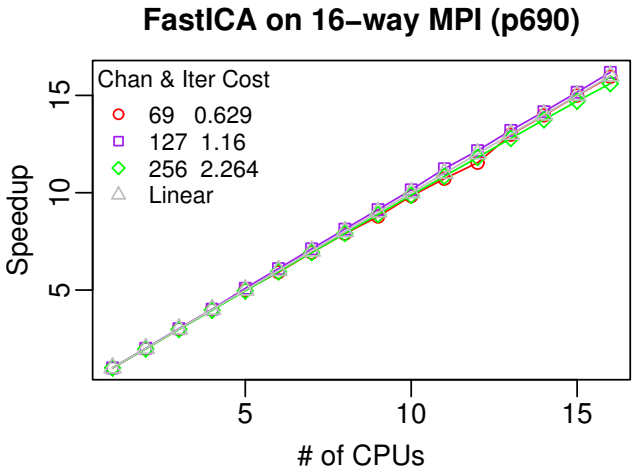
Because the FastICA/MPI and Infomax/OpenMP algorithms use different parallelization approaches, we treat these sets of measurements as incommensurate and describe each separately. In the graphs that de-

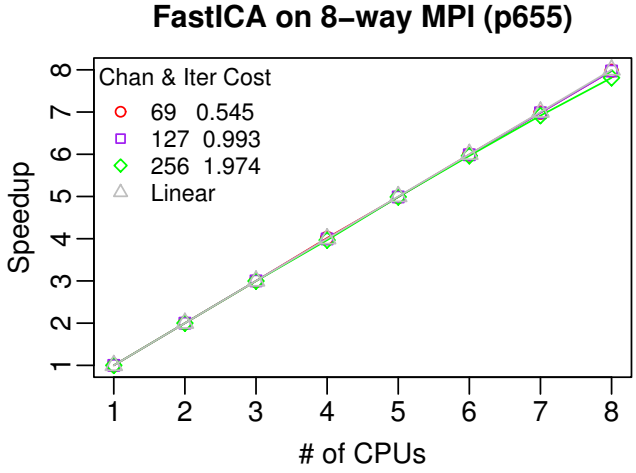
scribe *Speedup*, the speedup associated with n processors is the ratio of the cost-per-iteration for n processors to the cost-per-iteration for 1 processor. Table 2 summarizes the results displayed in these graphs. The speedup shown is maximum speedup obtained.

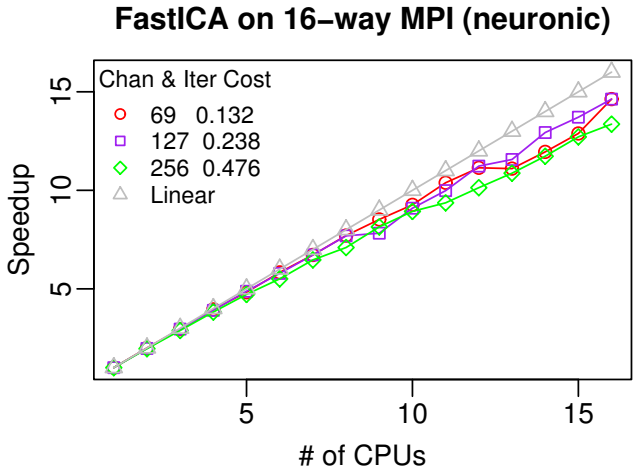
Table 2. Sequential Cost and Speedup for 69×111000 **data**

Hardware/Algorithm	C_{iter} (sec)	Speedup
G4 MATLAB FastICA	1.21	n/a
G4 FastICA	0.3089	n/a
Neuronic MATLAB FastICA	0.1974	n/a
Neuronic FastICA	0.1319	14.6296
P655 FastICA	0.5449	7.9768
P690 FastICA	0.6286	15.9566
G4 MATLAB Infomax	11.21	n/a
G4 Infomax	4.36	n/a
Neuronic MATLAB Infomax	5.22	n/a
Neuronic Infomax	1.0892	1
P655 Infomax	3.4366	3.26
P690 Infomax	3.9838	3.2519

The performance graphs demonstrate the speedup of the per-iteration cost for a given dataset when different numbers of processors are applied to the problem. Each curve corresponds to a different number of channels (the number of samples is fixed at 111,000 samples). The speedup for an n-processor computation is computed by dividing the per-iteration cost by the per-iteration cost for a single processor. The diagonal line represents linear speedup.

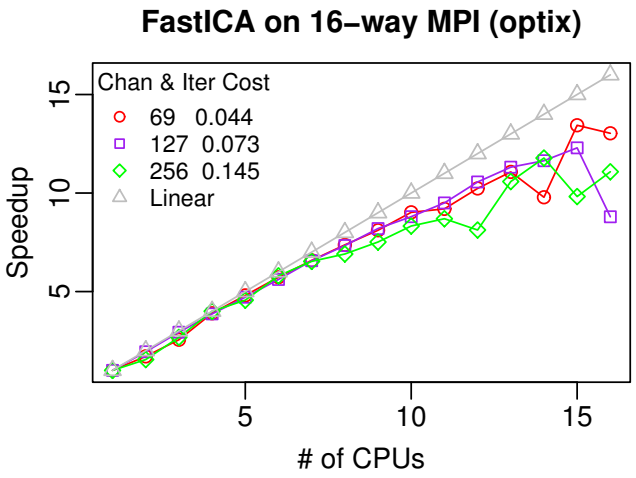


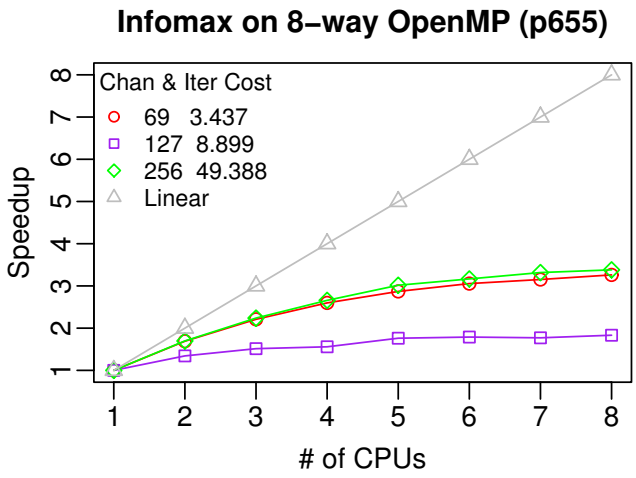


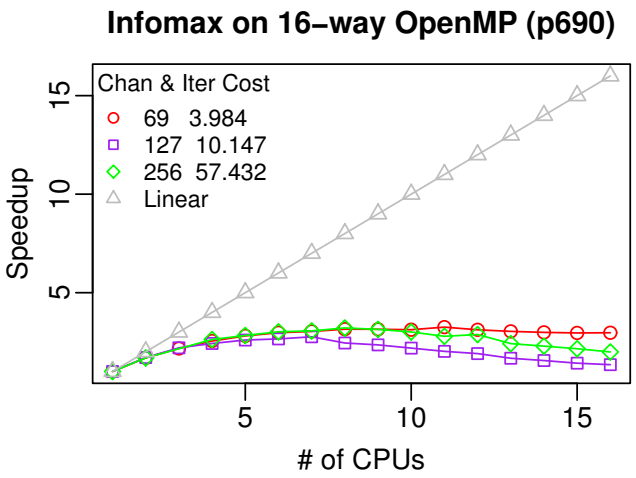


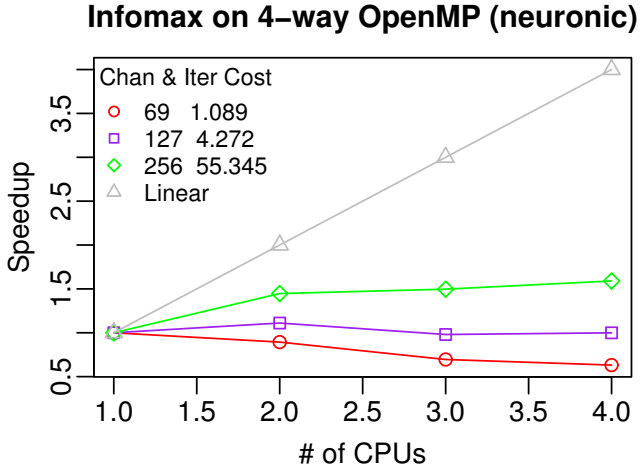
10. Expected vs. Observed Results

The FastICA algorithm equally partitions the samples among the available processors. An examination of the FastICA algorithm reveals only two significant points where synchronization and data exchange will occur between the various worker and head processes. Once for every desired component, where the initial weight vector for that component is exchanged; and once for every iteration of weight improvement, where the workers sum their weight estimates. All of the other work is executed in parallel by the workers, operating upon their own partition of the data. The computation to communication time ratio is high for FastICA, resulting in a high level of parallel performance. The FastICA performance graphs show near-linear speedup for the different platforms.





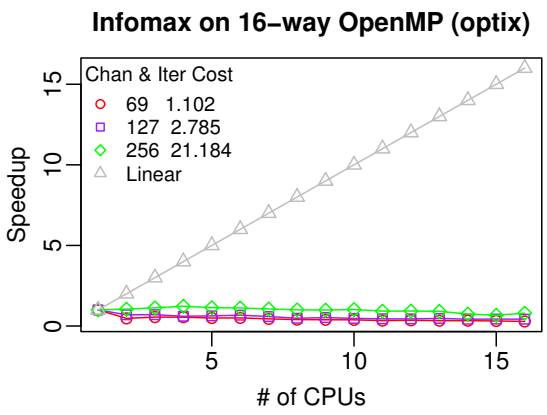




In contrast, the performance of the parallel Infomax algorithm is more constrained. Our analysis of the parallel Infomax algorithm shows several points where parallelism is occurring, during neural network training and the first part of updating the weights. We should see performance scaling for these sections. Unfortunately, the performance scaling curves indicate must poorer performance than FastICA. The reason is that the amount of parallel work available in OpenMP parallel regions is not large in relation to the sequential computation (being done in duplicate). In addition, the algorithm suffers caching effects due to the composition of blocks from random selection of samples. These blocks are then separately processes by OpenMP threads, but reference memory locations across the dataset. More interestingly, we see in the performance curves that 64-channel speedup improves better generally than 128- and 256-channel experiments, especially on the IBM p655. This is unexpected, but after analysis is explained by the fact that more of the data set can be contained in the Level 3 cache memory. For 64 channels and 110,000 samples, we suspect that the entirety of the dataset fits in the 32 Mbyte Level 3 cache on the IBM p655 machine. As the number of samples increase, we should see better speedups for larger numbers of channels.

11. Future Directions for HiPerSAT

The HiPerSAT library and tools currently provide ICA decomposition via the FastICA and Infomax algorithms. We plan to enhance the usability of HiPerSAT by increasing the performance of the existing algorithms, as well as by implementing additional ICA



methods that have shown promise in EEG analysis. In addition, we will be building grid-based interfaces to permit the remote initiation and monitoring of HiPerSAT tasks.

11.1. Performance

Both the FastICA and Infomax algorithms are very processor-intensive, especially with large data sets. The early results are promising and show the potential of exploiting parallelism for these tasks. However, there are several potential performance enhancements that bear investigating.

Although FastICA scales linearly with the number of processors, we can further improve overall performance by reading and writing the EEG data in parallel. This can be done directly during the FastICA execution by using MPI's asynchronous sends and receives to parallelize data distribution.

Clearly, the Infomax algorithm has room for performance improvement. Our scaling results show a best-case speedup of 3 running on 8 processors, and poorer performance overall. Although OpenMP performance is highly dependent on the platform and compiler, we need to further investigate ways to increase the portion of the algorithm that can operate in parallel. This includes minor changes such as adjusting the block size used during the training of a weight vector, and major changes such as allowing each worker thread to work on its own block in parallel, merging the learned weights after each step. The mathematical legitimacy of these optimizations must be analyzed.

As mentioned in 7.4, we have come up with a more memory-efficient way to sort and export the independent components from Infomax in a windowed fashion which does not require duplication of the data before output.

Finally, we want to consider alternative parallelism approaches, including the design of an MPI-based Infomax, an OpenMP-based FastICA, and a hybrid MPI/OpenMP version of both algorithms, assuming these versions offer opportunities for improved performance.

11.2. New ICA Features and Algorithms

The versions of FastICA and Infomax currently implemented in HiPerSAT lack some features that are present in the EEGLAB-based versions of these algorithms (`runica` and `fastica`). We plan on implementing these features within HiPerSAT. They include:

- For both FastICA and Infomax, the option of performing PCA (Principal Component Analysis) reduction during preprocessing
- Optional restarts with a randomized weight matrix to reduce the likelihood of a complete lack of convergence in FastICA.
- Performing *symmetric-mode* FastICA instead of *deflationary-mode*
- Performing *extended-mode* Infomax, which allows the detection of subgaussian components.

There are a variety of other ICA algorithms that have been conceived and implemented in MATLAB. These algorithms differ in their assumptions about the data, the way they measure independence, and whether they account for temporal ordering of the data. They include SOBI (Second-Order Blind Identification) [8], JADE (Joint Approximate Diagonalization of Eigen matrices) [4], and ERICA (Equivariant Robust ICA) [5]. We will be examining these algorithms to determine their utility in the neuroscience domain and their suitability for high-performance implementation.

11.3. Grid Execution

Our HiPerSAT implementations of ICA algorithms require that the `hipersat` program operate on a parallel system. However, a researcher may use MATLAB on their laptop or workstation to do much of their interactive analysis. We are extending the MATLAB integration to allow remote execution of HiPerSAT tasks on computational servers. Longer term this will utilize a grid-enabled interface where HiPerSAT will be accessible as a grid service. One important advantage that will come from this will be the ability to use HiPerSAT on multiple EEG data sets concurrently.

12. Conclusions

We described the ICA class of techniques and parallel implementations of two ICA algorithms, FastICA and Infomax. We showed that for the large data sets typically found in dense-array EEG research, the performance provided by HiPerSAT was significant compared to the sequential implementations in EEGLAB (which do employ fast LAPACK libraries in MATLAB). The speedup for FastICA was impressive while only modest for Infomax. The ability to handle dataset sizes larger than the MATLAB-based versions is very important for the EEG neuroimaging community at large.

The FastICA algorithm was shown to scale quite nicely, with near linear speedup as additional processors are added. However, the Infomax algorithm scaled sublinearly. This lack of parallelism and speedup in HiPerSAT Infomax is a consequence of the current implementation and warrants further research. However, one might ask whether this is even worth it given the outstanding showing of parallel FastICA. Although FastICA is faster and parallelizes better than Infomax, it has been shown that it can fail to converge on a solution. This appears to be especially problematic for larger numbers of channels (e.g., more than 127). Restarting the algorithm with new random weight vectors can reduce, but not eliminate, this convergence problem.

Although not the subject of this paper, the fact is that the choice of Infomax versus FastICA is also one of perceived correctness and quality of results. Neuroscientists will select among the various decomposition algorithms for reasons other than performance. Indeed, one active area of neuroscience research is the comparison of different ICA algorithms as applied to EEG. Different algorithms are appropriate for different assumptions about the EEG data and the underlying signals.

Currently, we are actively collaborating with neuroscience researchers at the University of Pittsburgh in the application of HiPerSAT in cognitive language processing studies focusing on word learning and reading assessment. The EEGLAB-based ICA implementation was deficient for this work because of the large datasets involved. HiPerSAT has also been successfully applied for artifact detection and cleaning in this and other neuroscience work. In general, we see the HiPerSAT library and tools as a direct replacement for the EEGLAB ICA implementations, providing the EEG/MEG community with higher performing, parallel solution.

13. Acknowledgment

The authors would like to thank Virat Agarwal, who provided valuable assistance in the development of HiPerSAT and the measurement of its performance. This research was partially funded by a grant from the National Science Foundation, Major Research Instrumentation program, and a contract from the Department of Defense, Telemedicine Advanced Technology Research Center, for the University of Oregon's Brain, Biology, and Machine Initiative Neuroinformatics Center.

References

[1] J. K. Aapo Hyvärinen and E. Oja. *Independent Component Analysis*. Wiley, 2001.

[2] E. e. a. Anderson. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] A. J. Bell and T. J. Sejnowski. An information-maximisation approach to blind separation and blind deconvolution. *Neural Computation*, pages 1129–1159, 1995.

[4] J.-F. Cardoso and A. Souloumiac. Blind beam-forming for non gaussian signals. In *IEE Proceedings-F*, pages 362–370, 1993.

[5] J.-F. Cardoso and A. Souloumiac. Robust blind source separation algorithms using cumulants. In *Neurocomputing*, pages 87–117, 2002.

[6] A. Delorme and S. Makeig. EEGLAB: an open source toolbox for analysis of single-trial EEG. *Journal of Neuroscience Methods*, 134:9–21, 2004.

[7] EEGLAB home page, 2005.

[8] A. B. et al. A blind source separation technique using second order statistics. In *IEEE Trans. on Signal Processing*, pages 434–444, 1997.

[9] L. S. B. et al. An updated set of basic linear algebra subprograms (blas), 2002.

[10] S. M. et al. Mining event-related brain dynamics. *Trends in Cognitive Science*, pages 204–210, 2004.

[11] W. G. et al. A high-performance, portable implementation of the mpi message passing interface standard.

[12] R. D. Greg Burns and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[13] Cocktail party demo, 2005.

[14] HUT - CIS: The FastICA package for MATLAB, 2005.

[15] A. Hyvärinen. A fast fixed-point algorithm for ica. *Neural Computation*, pages 1483–1492, 1997.

[16] A. Hyvärinen. Fast and robust fixed-point algorithms for ica. *IEEE Transactions on Neural Networks*, pages 626–634, 1999.

[17] Openmp application program interface.

[18] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, pages 379–387, 2003.

[19] J. V. Stone. *Independent Component Analysis : A Tutorial Introduction*. MIT Press, 2004.
[20] TAU home page, 2005.
[21] D. Tucker. Spatial sampling of head electrical fields: the geodesic sensor net. *Electroencephalography and Clinical Neurophysiology*, pages 145–163, 1993.