Parallel Memory-Efficient Processing of BCI Data

Trevor Alexander*, Anthony Kuh*, Katsuhiko Hamada†, Hiromu Mori‡, Hiroyuki Shinoda†, Tomasz Rutkowski‡
*Department of Electrical Engineering, University of Hawai'i at Mānoa, Honolulu, Hawai'i, USA
E-mail: trevor@bci-lab.info
†The University of Tokyo, Tokyo, Japan
‡Life Science Center of TARA, University of Tsukuba, and RIKEN Brain Science Institute, Japan
E-mail: tomek@bci-lab.info

***Abstract*—Following after magnetic resonance imaging (MRI) and electrocortigraphy (ECoG), electroencephalography (EEG)-based research is entering the world of big data[1]. A research-quality brain-computer interface (BCI) data set can easily number in the hundreds of millions of points, making methodology of processing and classification critical. A selection of broadly applicable optimization methods implemented in R is presented that enables users to take advantage of parallelization, guaranteed call-by-reference to limit memory overhead, and scalable performance with common BCI processing tasks. As proof of concept, classification results for a P300 experiment and performance statistics are presented.**

I. Introduction

A host of different groups are at work on improving the processing of large data sets for machine learning and classification [2][3][4][5][6]. One such group identifies R as "ideal" for expressing machine learning algorithms for high performance computing, and is at work on a distributed incremental processing framework [2]. However, no work has been done (to our knowledge) to explicitly improve processing of EEG data sets using high performance computing (HPC) methods. As more (computationally) elaborate processing and classification methods are introduced, the time and space required for initial training and actual use become increasingly critical. Despite this, consideration of these factors does not have to be postponed when testing or evaluating a proposed algorithm for feasibility or relative effectiveness.

In this paper, we discuss practices and approaches for accelerating BCI data processing. We begin with a discussion of a flexible layout for EEG data, and introduce a fast and efficient data structure that supports this layout. Next, we define an example workflow from filtering to classification, and show competitive classification results on actual data. For selected sections in the workflow, we describe two easily applicable optimization methods for drastically reducing the time and space required for processing, namely parallelization and reduction of memory overhead through enforcement of call-by-reference. We demonstrate their improvement through benchmarking. We conclude by noting the significant performance increases obtained and discuss future work in this vein.

II. Organizing and Structuring Data

A. Principles

EEG must be structured to serve different needs, depending on the method of analysis being applied. This results in a number of competing formats–for example, BCI2000 (DAT) [7], Emotiv (EDF) [8], BioSemi (BDF) [9], and EEGLAB/-MATLAB (MAT) [10] all use entirely different formats each with different structure to store their data. This issue is typically left out of most discussions in the fields of machine learning and statistical analysis, but is responsible for a major reduction in efficiency–as much 80% of time spent on data analysis and modeling is spent modifying the formatting or layout of the data, according to one estimate[11].

In an attempt to avoid meaningless proliferation of *ad hoc* formats and *de facto* "standards", we adhere as much as possible to the following principles:

- Flexibility: the structure must accommodate multiple use cases with minimal syntactic effort
- Efficiency: the structure must be quickly accessible and compact in memory (or on disk)
- Transparency: the structure must be *understandable* and natural to the researcher or programmer

B. Format

The table layout; i.e., the row-column ordered representation of data, is now the most ubiquitous container format in computing, statistical analysis, and data science. This representation appears most notably in databases, whose implementations have become omnipresent. This trend holds for time-series data, of which EEG is a member. Thus, the fundamental data structure for EEG data should also be a table. Placing EEG data in such a container allows the logic of existing libraries and toolboxes in these areas to be immediately applied.

At the same time, the structure must also support "matrix-type" operations without recourse to conversion, otherwise the effort is wasted. The query language of table structures supports this very well, allowing for subsetting and grouping in a transparent, understandable manner. In this way, we adhere to the above principles in a standard form.

Specifically, we make extensive use of `data.table` [12], a high-performance data structure available in R, for the vast majority of tasks in the analysis workflow[1]. In the following

[1]R was chosen over Python or MATLAB for the following additional reasons: parallelization is available through freely available packages with modular support for a wide variety of backends (both local and distributed), and for its strong academic machine learning community, which makes available peer-reviewed packages for free (attributed) use.

978-616-361-823-8

TABLE I
"LONG" FORM OF EPOCHED EEG DATA

Sample	Trial	Class	Channel	Voltage	Time
1	1	0	1	-14.36	-100
1	1	0	2	-15.94	-100
1	1	0	3	-22.85	-100
1	1	0	4	-47.43	-100
1	1	0	5	-65.98	-100
⋮	⋮	⋮	⋮	⋮	⋮

TABLE II
"WIDE" CHANNEL FORM OF EPOCHED EEG DATA

Class	Time	Trial	Ch1	Ch2	Ch3	…
0	-100	4	0	0.00	0.00	…
0	-100	47	0	0.53	0.00	…
0	-100	96	0	0.00	0.00	…
0	-100	174	0	0.00	0.00	…
0	-100	205	0	0.00	-0.11	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮

section we give a detailed explanation of the layout and use.

C. Layouts

Conceptually, a table is most simply stated as a structure that is "all columns." When we say this, we mean that all identifiers are columns–the sequence of rows is dependent on the ordering of values in a given column. In order to make the concept more clear, we provide an example of an epoched EEG data set in the above format in Table I.

As you can see, all possible dimensions are unrolled into column indices, such that a unique tuple of indices specifies a unique `Voltage`, our value of interest. This is known colloquially as a "long" table format [11], and is standard for SQL and other database types.

While the "long" format is the most flexible layout, matrix operations on multiple channels are common enough that a "wide" channel format is useful. The "wide" format can be quickly generated. An example is given in Table II.

With the above two layouts, we can accomplish every task needed for basic EEG-BCI analysis and classification without storing the data in any other form. This keeps both code clean and memory usage tight, as we will see in the following sections.

III. EXPLORATORY ANALYSIS

A. Experiment

For this analysis, we use P300 experiment data collected from a novel tactile BCI experiment[13]. Notably, stimulus was presented using contactless airborne ultrasound (airborne ultrasonic tactile display, or AUTD). Data was sampled at 512Hz. The stimulus length and interstimulus interval were both 400ms. Initial processing included a bandstop filter from 48-52Hz for power line interference, and a bandpass filter from 0.1-60Hz. Thirteen male subjects participated in three experiment sessions, which each contained a randomly ordered set of 90 targets and 450 non-targets. The resulting time series was epoched by channel into separate trials, over the interval -100 to 900 milliseconds (pre- and post-stimulus, with overlapping samples duplicated).

B. Process and Functions

1) Filtering: We further applied a Butterworth lowpass filter with cutoff frequency of 10Hz (backwards and forwards to avoid disturbing signal phase) to the epoched signals. The magnitude response is given in Figure 1. For reference, plots of the unfiltered and filtered average spectra follow in Figures 2 and 3.

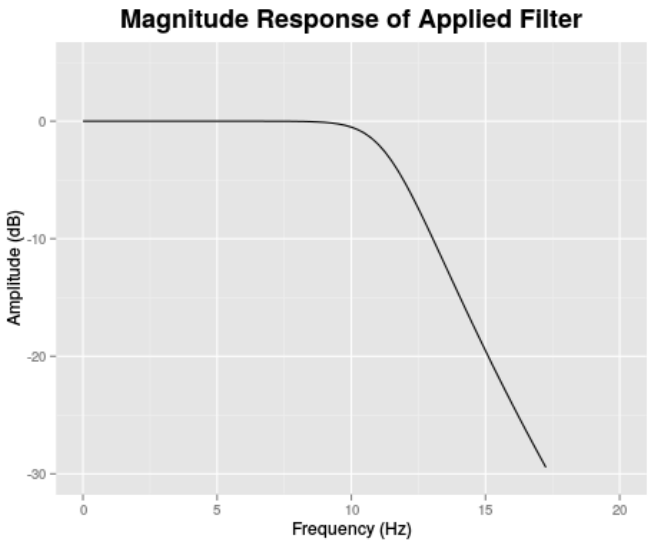

Fig. 1. Low pass filter applied to each channel and trial.

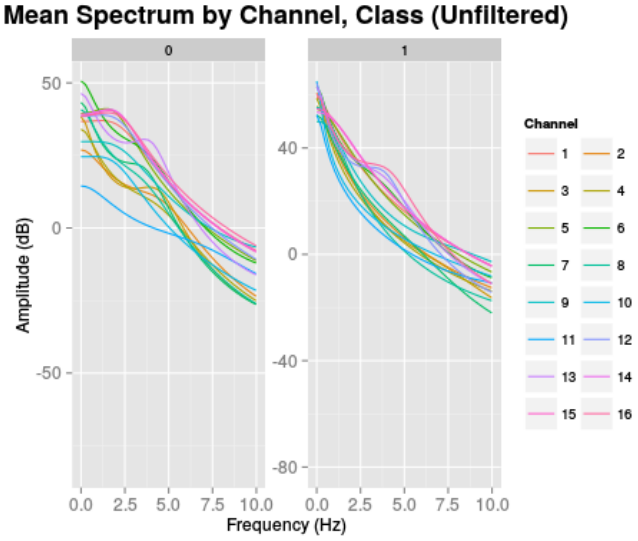

Fig. 2. Unfiltered spectrum of averaged trials by class.

The resulting signal was also downsampled by a factor of approximately 6 in order to reduce memory demands and increase processing speed.

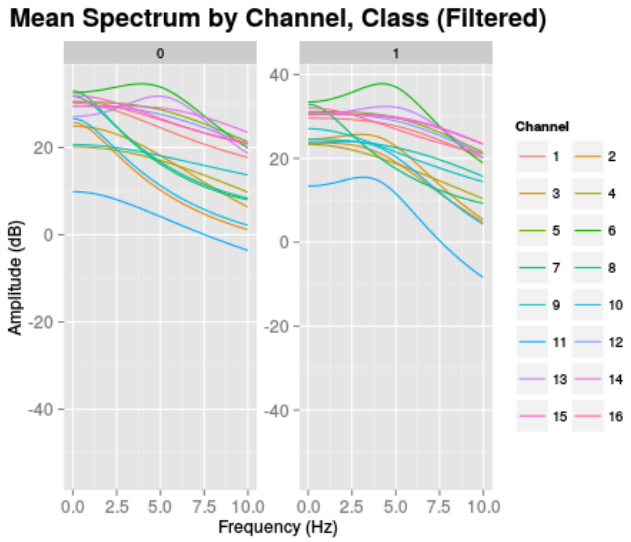

Fig. 3. Unfiltered spectrum of averaged trials by class.

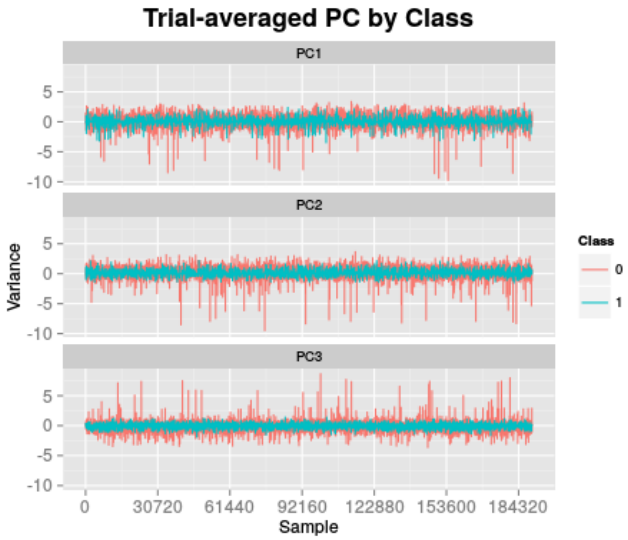

Fig. 5. Karhunen-Loeve transformed training set ordered by sample and trial (first 3 components).

2) Principal Component Analysis: Following filtering and downsampling, EEG data was separated by subject and session into two training sessions and one testing session. Additionally, only the subset of data from 200-450ms was considered, resulting in 16 channels of 22 samples each, for each trial. Principal component analysis was performed on the training set using SVD for efficiency [14].

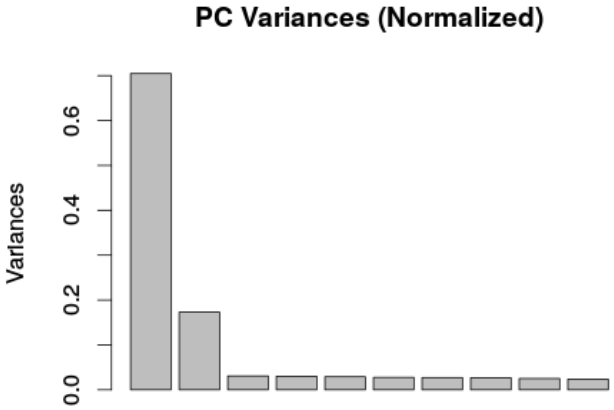

Fig. 4. Principal component variances of training data (subject 13 pictured).

As shown in Figure 4, the vast majority of the variance is explained by the first two principal components (we also include the third component for illustrative purposes).

As we can see from Figure 5, the training set samples appear to be well separated along the different principal component axes.

IV. Classification

A. Process and Results

As each subject performed the same experiment for three separate sessions, we used the first two sessions to train each classifier, and held the third session back for testing. Features were chosen from the Karhunen-Loeve transform (derived from the first three principal component rotations of the training set) of the 22 samples (in each of 16 channels) in the time interval from $200 - 450$ ms. Since class proportions were unequal (a 5:1 ratio), the training set was resampled with replacement to ensure learners were trained on equal proportions. Additionally, uniform priors were provided to each learner (equal class weights in the case of SVM), and the confusion matrix was inspected for error bias.

Two learners were tested: linear discriminant analysis with shrinkage (SDA) [15], and a linear support vector machine (L_2-regularized with L_1 loss) [16]. Each learner was trained on the first two sessions with each subject, and tested on the third session over 10 different runs.

Results are shown in Table III.

V. Optimizations

Our optimization efforts were focused in two areas: parallelization and reduction of memory use.

Parallelization is aimed foremost at *scalability*. Parallelization has been mostly ignored in the BCI literature, and the most popular toolbox, EEGLAB, does not explicitly provide for parallelization in its code [17]. Given the increase in EEG data set size and future proliferation thereof (such as that proposed by the Neural Engineering Data Consortium [1]), scalability is critical, as indicated by Gustafson's law [18].

Gustafson's law (illustrated in Figure 6 states that given the "scaled speedup" or ratio of the serial process runtime to

TABLE III
ACCURACY OF TESTED CLASSIFIERS BY SUBJECT

Subject	SDA	SVM
1	$75.9\% \pm 0.000\%$	$82.9\% \pm 0.553\%$
2	$76.0\% \pm 0.000\%$	$81.6\% \pm 1.51\%$
3	$66.0\% \pm 0.000\%$	$80.9\% \pm 1.26\%$
4	$74.1\% \pm 0.016\%$	$82.2\% \pm 0.324\%$
5	$74.4\% \pm 0.173\%$	$81.5\% \pm 1.48\%$
6	$76.7\% \pm 0.000\%$	$82.4\% \pm 0.841\%$
7	$71.5\% \pm 0.332\%$	$81.4\% \pm 1.83\%$
8	$72.2\% \pm 0.107\%$	$82.2\% \pm 0.665\%$
9	$80.5\% \pm 0.010\%$	$81.9\% \pm 2.06\%$
10	$72.5\% \pm 0.032\%$	$82.1\% \pm 1.36\%$
11	$77.9\% \pm 0.002\%$	$82.3\% \pm 0.538\%$
12	$75.6\% \pm 0.013\%$	$82.2\% \pm 1.28\%$
13	$65.8\% \pm 0.023\%$	$81.0\% \pm 1.33\%$
Average	73.78%	81.89%

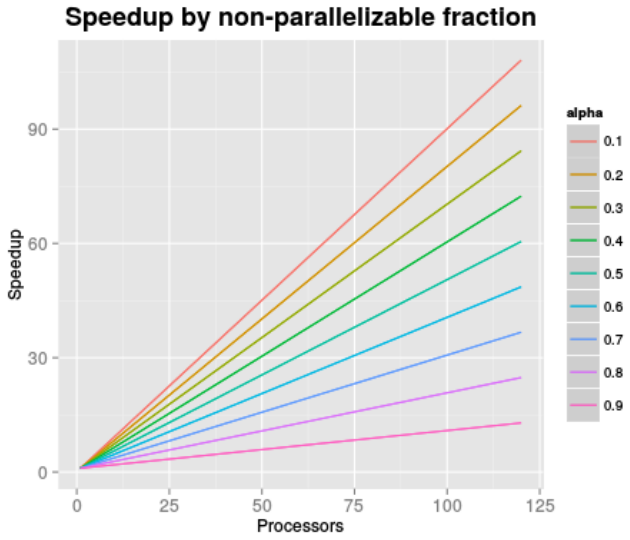

Fig. 6. Illustration of expected speedup with increasing cores for different values of α according to Gustafson's law.

the parallel runtime $S(P)$, where P is an integer number of processors, the following relation holds:

$$S(P) = P - \alpha(P - 1) \tag{1}$$

Here, α represents the serial fraction of the process, that is, the fraction that cannot be parallelized. In other words, the law describes the scalability of a calculation–it answers the question, "How much can we get by throwing more power at this process?"

The other main dimension determining processing performance is occupied memory. While memory is inexpensive and analysis can be offloaded (at least in the research phase) to cloud computing grids like Amazon's EC2, this is no justification for inefficiency. Considering that one of the goals of BCI is a complete system that can be used by a patient independent of a general-purpose computing platform, (i.e., an *embedded* system), the issue merits consideration. Our main technique was to ensure copying is minimized by guaranteeing call-by-reference in all possible cases.

A. Parallelization

Parallelization in R consists of two components: a frontend that dispatches and collects calculation tasks as jobs, and a backend which manages the (R) processes that perform the jobs. This allows for modularity–depending on the available hardware, the same code can be used on a single multicore machine or a cloud computing cluster.

In this workflow, we use the frontend `foreach`[19] to parallelize iterative tasks, and `doMC`[20] to perform these tasks on multiple cores of a single machine.

1) Filtering: Filtering of epoched EEG trials is an *embarassingly parallel* process [21]. By this we mean that the function runs on separate sections of data independently, making it a perfect candidate for parallelization.

Parallelization in this workflow is done explicitly through the `foreach` package [19], which provides a simple framework for dispatching iterative tasks to a parallel backend. In this implementation we use `doMC` to distribute tasks to each core. Other backends are available, including an interface to MPI (message passing interface)-compatible clusters [22].

2) Hyperparameter optimization: Hyperparameter optimization is another embarrassingly parallel task. While grid search is the most inefficient of all methods, it is used here as a simple illustration. We performed a grid search over the two parameters λ (shrinkage intensity for the correlation matrix) and λ_{var} (shrinkage intensity for the variances) for shrinkage LDA [15], using the same microbenchmark suite as in the filtering task. Results for this grid search are given in Figure 7.

Listing 1. Benchmarked expression for SDA hyperparameters.

```
foreach(this.lambda=iter(subset$lambda),
        .combine=cbind) %:%
 foreach(this.lambda.var=iter(subset$lambda.var),
        .combine=rbind) %dopar% {

    sda.model <-
        sda(lambda.freqs = 1,
            lambda=this.lambda,
            lambda.var=this.lambda.var,
            Xtrain=as.matrix(training.pca[,1:3,
                          with=FALSE]),
            L=training.pca[,as.factor(Class)])

    sda.pred <-
        predict(
         sda.model,as.matrix(testing.pca[,1:3,
                          with=FALSE]))

    acc <- sum(sda.pred$class !=
               testing.pca[,Class])
           /testing.pca[,length(Class)]

            if (acc < 0.5) { acc <- 1 - acc}

            acc
 }
```

The benchmark results for SDA with increasing parallelism are shown in Figure 8. The same benchmark was performed

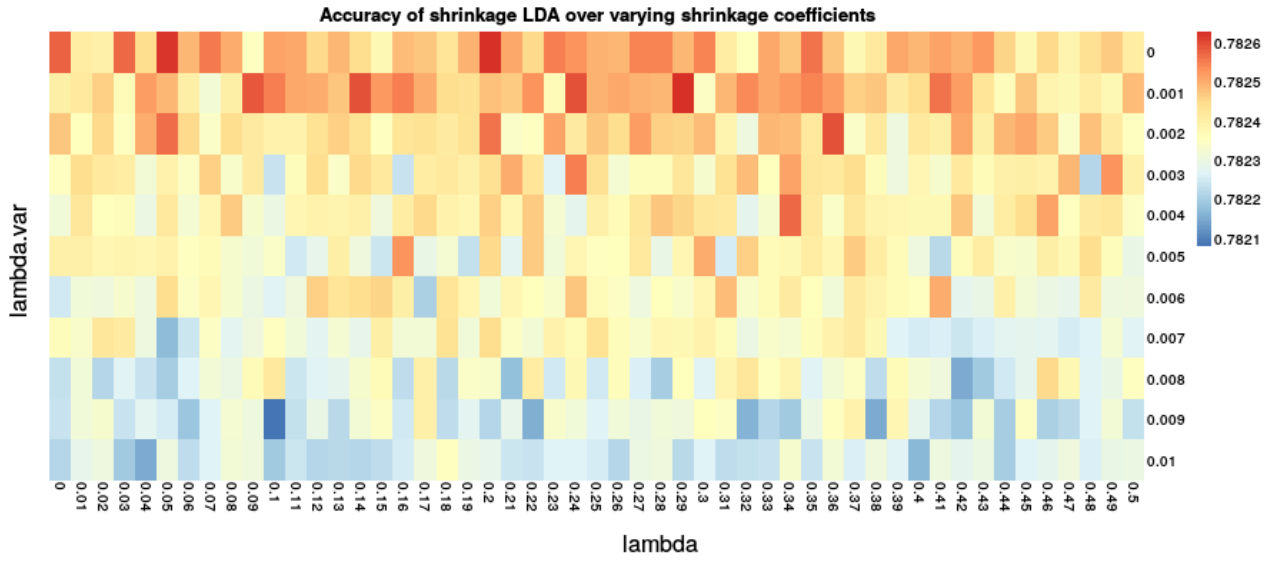

Fig. 7. SDA accuracy over a subset of two hyperparameters for a single subject (1).

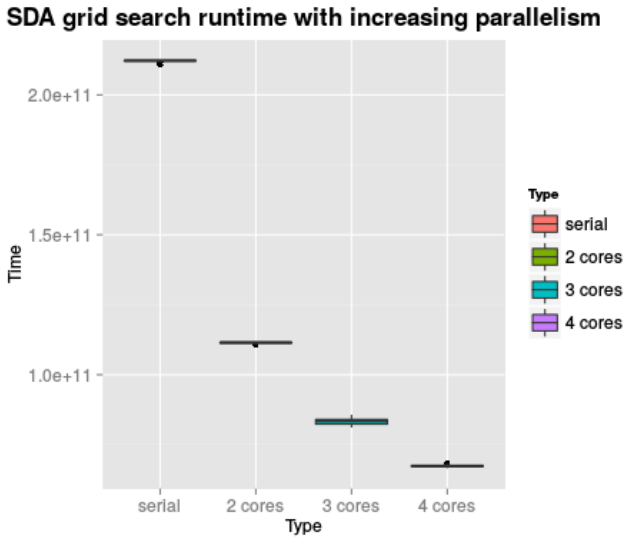

Fig. 8. Box and whisker plot of SDA hyperparameter grid search runtime with increasing parallelism.

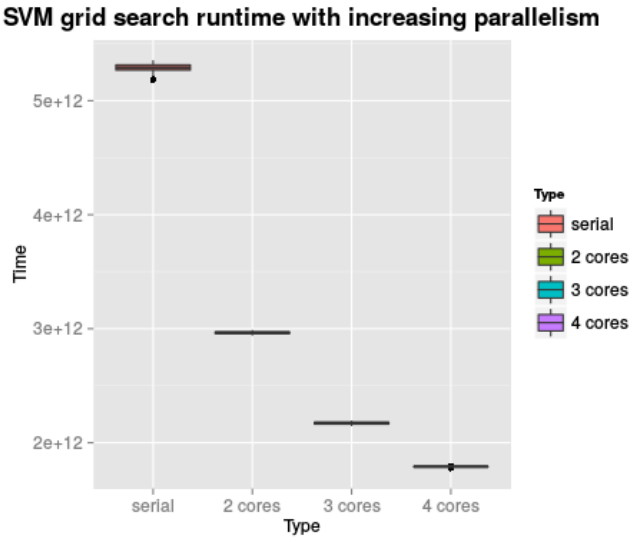

Fig. 9. Box and whisker plot of linear SVM hyperparameter grid search runtime with increasing parallelism.

using the linear SVM classifier (for brevity, we omit similar code here), shown in Figure 9.

B. Memory optimization

Both R and MATLAB, by default, pass objects to functions in a "lazy" manner–they do not copy objects in memory unless accessed within the current scope [14][23]. While this is an effective general approach, it relies wholly on the programmer to implement functions so that the changes are isolated to the smallest possible subset of whatever variable is passed. A better approach is to guarantee function passing by reference. This can be directly achieved with minimal effort using `data.table`. For example:

Listing 2. Windowed average using call-by-reference.

```
avg.with.group <-
   input.table[,roll_mean(get(val.col),
               n = window.width),
               by=c(id.cols)]
```

The above code computes a windowed average of one column in the table[24][25], grouped by any subset of other columns (such as Channel or Trial), using no more memory at any point in time than that required to store the number of elements in the window. It is also extremely concise and relatively readable.

TABLE IV
BENCHMARK SYSTEM SPECIFICATIONS

CPU	Intel Xeon E3-1225 V2 @ 3.20GHz (4 cores)
Memory	32GiB DIMM DDR3 Synchronous @ 1333MHz
OS	Debian Linux (kernel version 3.12-1-amd64)
R	version 3.1.0

TABLE V
FILTERING RUNTIMES WITH INCREASING PARALLELISM

Type	min	median	max
serial	59.82971	60.0070	60.53914
parallel, 2 cores	56.01123	57.2282	57.77827
parallel, 3 cores	44.92327	45.61265	46.04711
parallel, 4 cores	39.16282	41.14741	41.58670

The salient feature here is that `data.table` allocates only as much memory as is strictly needed to execute the function given within the brackets, on the columns named there and within the `by` grouping. MATLAB users may think of this as a further "boost" to the concept of vectorized code–with `data.table`, users do not have to explicitly form the subsetting or mapping vectors mentally for each function they wish to speed up–they simply name the grouping they wish to operate on and any ordering of rows by setting a key (which further speeds up large operations). Thus, memory usage is strictly reduced and syntax is simplified.

VI. BENCHMARKS

All benchmarking and processing was performed on a Lenovo ThinkServer TS130. Detailed specifications are given in Table IV.

A. Time

1) Filtering: Parallel and serial processing was compared by filtering 100 million data points from a real epoched EEG data set with a 10Hz Butterworth filter. This process was repeated 20 times with 1 core, 2 cores, and 3 cores and microbenchmarked[26]:

Listing 3. Benchmarked filtering expressions.

```
filter.seri <-
    expression(erp[1:100000000,
     filtfilt(low.pass,
     by=c("Trial","Channel")])

filter.para <-
    expression(erp[1:100000000,
     foreach(
      this.trial=iter(erp[,unique(Trial)]),
      .combine=c) %dopar% {

      filtfilt(low.pass,erp[J(this.trial),
      Voltage])}])
```

From the results shown in Figure 10, we may roughly estimate the non-parallelizable fraction α (the portion that is responsible for the nonlinear decrease in time) of the filtering process using Gustafson's law[18].

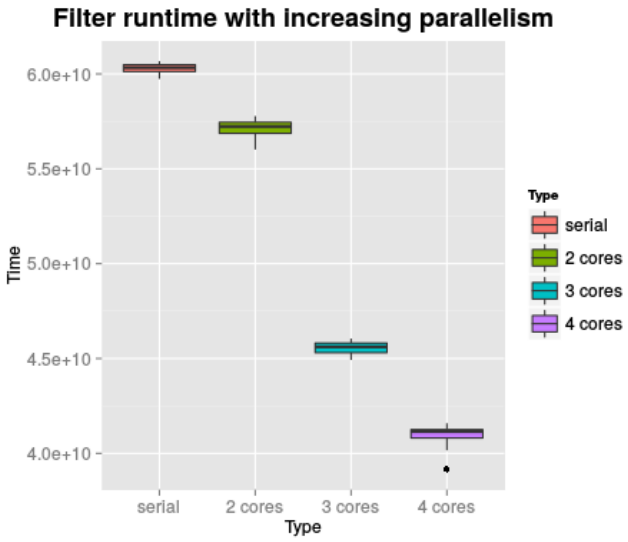

Fig. 10. Box and whisker plot of filter runtime with increasing parallelism.

TABLE VI
ESTIMATED PARALLELIZABLE FRACTION OF FILTER PROCESS

$S(P)$	P	α	parallel fraction
1.049	2	0.951	4.9%
1.316	3	0.842	15.8%
1.458	4	0.847	15.3%

Thus we can expect that about 15% of the filtering process is parallelizable on the testing machine (and its corresponding overhead). Under this estimate, the same 100 million data points can therefore be filtered in roughly 10 seconds on a relatively small 32-core cluster.

2) Classifier hyperparameter search: Similar trends appear when comparing learner performance over ranges of hyperparameters. For each learner, we measured the learner accuracy over the chosen hyperparameter subsets. For shrinkage SDA, we searched over subsets of shrinkage intensity for the correlation matrix `lambda` and shrinkage intensity for the variances `lambda.var`. For linear SVM, we searched over subsets of the cost.

TABLE VII
SDA RUNTIMES WITH INCREASING PARALLELISM

Type	min	median	max
serial	210.72354	212.20380	212.40739
parallel, 2 cores	100.76444	111.45896	111.86055
parallel, 3 cores	81.09276	83.45086	85.63844
parallel, 4 cores	66.55676	67.30343	68.41378

TABLE VIII
LINEAR SVM RUNTIMES WITH INCREASING PARALLELISM

Type	min	median	max
serial	5182.834	5288.831	5315.910
parallel, 2 cores	2939.684	2965.018	2989.479
parallel, 3 cores	2146.640	2172.414	2193.440
parallel, 4 cores	1776.822	1790.169	1799.507

TABLE IX
ESTIMATED PARALLELIZABLE FRACTION OF SDA HYPERPARAMETER SEARCH

$S(P)$	P	α	parallel fraction
1.904	2	0.096	90.4%
2.543	3	0.229	77.1%
3.153	4	0.282	71.8%

TABLE X
ESTIMATED PARALLELIZABLE FRACTION OF LINEAR SVM HYPERPARAMETER SEARCH

$S(P)$	P	α	parallel fraction
1.784	2	0.216	78.4%
2.435	3	0.283	71.7%
2.954	4	0.349	65.1%

Runtimes are shown in Tables VII, VIII. As seen in Tables IX and X, the parallelizable fraction is quite high, at least 65%. This is expected, since the parameter search process is entirely iterative.

B. Memory

Our primary statistic for measuring performance is the peak ratio of bytes of memory used in calculation (as measured by the amount of memory released afterward), to the size in bytes of the original input data.

Three processes were profiled for memory use: grand trial mean of all recorded signals, a windowed time average (included for the sake of completeness) over each trial, channel, and subject, and low pass filtering of all trials and channels. For efficiency's sake, the initial input data was constructed from only the first 100 trials from each subject, for a total size of 442.3 megabytes (SI). Results are collected in Table XI, and the calls benchmarked follow below.

1) Call types: To provide a call-by-value comparison with `data.table` while preserving concise grouping calls, the function `ddply` was used [27] . Its syntax is similar and provides a good comparison, as it executes essentially the same procedure as the `data.table` calls. Each of the below code listings contains a `data.table` implementation and `ddply` implementation of the same task, used with the memory profiler.

Listing 4. Memory benchmark (grand mean).
```
mean.by.ref <-
        test.table[,mean(Voltage),
             by=c("Sample","Channel","Class")]

mean.by.val <-
        ddply(test.table, .(Sample,Channel,Class),
             summarize, meanv=mean(Voltage))
```

TABLE XI
MEMORY RELEASED BY FUNCTION (OVERHEAD)

Task	call-by-reference	call-by-value
EEG mean by trial	0 Mb (0%)	2393.2 Mb (541%)
EEG windowed time average	0 Mb (0%)	2993.8 Mb (677%)
EEG low pass filter	948.44 Mb (214%)	2027.18 Mb (458%)

Listing 5. Memory benchmark (10-width windowed time average).
```
window.avg.by.ref <-
    test.table[,roll_mean(Voltage, n = 10),
           by=c("Subject","Trial","Channel")]

window.avg.by.val <-
    ddply(test.table, .(Subject, Trial, Channel),
           summarize,
           wavg = roll_mean(Voltage, n=10))
```

Listing 6. Memory benchmark (low pass filter).
```
end.f <- 20
start.f <- 10
low.pass <-
   butter(buttord(Wp = start.f/sample.rate*2,
                  Ws = end.f/sample.rate*2,
                  Rp = 0.5, Rs = 40))

filt.by.ref <-
  test.table.filt[,Voltage:=filtfilt(low.pass,
                                     Voltage),
                   by=c("Trial","Channel")]

filt.by.val <-
  ddply(test.table.filt, .(Trial, Channel),
       mutate,
       Voltage=filtfilt(filt=low.pass,
                        x=Voltage))
```

While frontends like `foreach` attempt to allow child processes to share as much memory allocated by the parent as is prudent, there is unavoidable overhead. In other words, blindly parallelizing every calculation is not a panacea. Inefficient memory use, even with well-written code, can be as high as seven times the occupied memory, as shown in table XI. When memory is at a premium, one workable strategy is to try to optimize as needed by structuring the data so that the calculation runs quickly even in a serial manner.

C. Real Time/Memory Effects

Processing speed is highly dependent on available hardware, but we can provide a mapping that relates the above time and memory savings to an actual experiment. Assume that a BCI processing chain consists of the following (memory/time independent) steps, for which the required time and memory are both unity:

1) Load data (w)
2) Filter data (x)
3) Preprocess/transform data (y)
4) Tune and verify classifier (z)

We can then describe the required time and memory mathematically by

$$t_w + t_x + t_y + t_z = 1 \tag{2}$$

$$\max\left(m_w + m_x + m_y + m_z\right) = M \tag{3}$$

where t_w is the required proportionate time for task w, m_w is the required memory for task w, and M is the total memory required. (They are hardware-specific.)

1) Time: As found above, the non-parallelizable fraction of the filtering task x is .880, while the non-parallelizable fraction of the classification task z for a linear SVM is 0.283. In practice, we found that the time required to compute the Karhunen-Loeve transform was comparatively negligible. Additionally, the time to load the data comes from a single fixed step, and can also be neglected in many cases. Thus, assuming no limit on available computing grid size (a reasonable assumption today), the time required to process the experiment given in this paper is:

$$t_{\text{parallel}} = 0.88t_x + 0.283t_z \tag{4}$$

If, say, the filtering step takes 10 percent of the total processing time, while hyperparameter tuning takes 90 percent, $t_{\text{parallel}} = 34.3\%$. In other words, if processing took 60 minutes on a laboratory's computing hardware with a non-parallelized program, this processing could be cut down to 24 minutes. The experiment introduced in this paper contains data processed independently from 13 subjects. In other words, if the data were available in advance, a BCI could be tuned for a subject in $\frac{24}{13} = 1.85$ minutes.

2) Memory: The case for memory is slightly different. Either there is enough memory available for processing, or there is not (virtual memory may be allocated from the hard drive, but this severely affects the processing time, and is only a fixed increase in space, which only kicks the ball down the road). In other words, it is the maximum of the memory requirements from each independent step that determines whether or not the process can complete.

In practice, we found that memory is only an issue through the transformation step–once the data is transformed and reduced in dimension, the problem is typically tractable. This means that the initial data size in memory and the overhead required for filtering are the measures of interest here. We assume that there is enough memory to load the data to begin with.

As noted in Table XI, the overhead required for a simple filtering process was 948.44 Mb, or 214% overhead, when call-by-reference was enforced through programming methods shown previously. On the other hand, call-by-value processing required an overhead of 2027.18 Mb, or 458% overhead, roughly 2.14 times as much. While circumstances vary widely with experiment and hardware available, it seems prudent to avoid having to double available memory, an upgrade that may necessitate further hardware upgrades to accomplish [2].

VII. Conclusions and Future Work

We have introduced a number of libraries and optimization methods in R taking advantage of high performance computing methods that can be used to accelerate processing and reduce memory usage of BCI algorithms. Through these improvements, significant speedups and reductions in memory use were shown through benchmarking. The workflow was validated using actual experimental data, and competitive results were obtained.

In a multi-core, cloud computing world, researchers should make the most of parallelization and optimization. High-performance computing methods do not just improve existing code–they also broaden the scope of what can be developed and tested. In order to make these and other improvements accessible to BCI researchers in a variety of disciplines, we are currently developing a full-featured EEG processing library in R with a graphical user interface. We are also building an R interface for streaming online data from BCI2000.

[2] i.e., a motherboard and CPU upgrade.

Acknowledgment

We thank the many library developers and R gurus on StackOverflow for their patient advice on numerous occasions, without which this paper would not be possible.

References

[1] I. Obeid and J. Picone, "Bringing big data to neural interaces," in *Proceedings of the Fifth International Brain - Computer Interface Meeting*, 2013.

[2] S. Venkataraman, I. Roy, A. AuYoung, and R. S. Schreiber, "Using R for iterative and incremental processing," 2012.

[3] O. Terzo, P. Ruiu, E. Bucci, and F. Xhafa, "Data as a service (DaaS) for sharing and processing of large data collections in the cloud," in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*, Jul. 2013, pp. 475–480. DOI: 10.1109/CISIS.2013.87.

[4] M. Paracha, S. Mohammad, P. Macfarlane, and J. Jenkins, "Real-time custom processing and delivery of large ecg data sets," in *Computers in Cardiology, 2003*, Sep. 2003, pp. 411–412. DOI: 10.1109/CIC.2003.1291179.

[5] H. Lin, S. Yang, and S. Midkiff, "RABID – a general distributed R processing framework targeting large dataset problems," in *Big Data (BigData Congress), 2013 IEEE International Congress on*, Jun. 2013, pp. 423–424. DOI: 10.1109/BigData.Congress.2013.67.

[6] T. Jejkal, V. Hartmann, R. Stotzka, J. Otte, A. Garcia, J. Van Wezel, and A. Streit, "LAMBDA – the LSDF execution framework for data intensive applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, Feb. 2012, pp. 213–220. DOI: 10.1109/PDP.2012.69.

[7] G. Schalk and J. Mellinger, *Technical reference: BCI2000 file format*. [Online]. Available: http://www.bci2000 . org / wiki / index . php / Technical_Reference : BCI2000_File_Format.

[8] B. Kemp and J. Olivan, "European data format 'plus' (EDF+), an EDF alike standard format for the exchange of physiological data," *Clin Neurophysiol*, vol. 114, no. 9, pp. 1755–1761, Sep. 2003.

[9] T. van Beelen, *Bdf+ format description*, Oct. 2012. [Online]. Available: http://www.teuniz.net/edfbrowser/bdfplus%20format%20description.html.

[10] *MATLAB(r) MAT-file format*, MathWorks, Inc., Mar. 2014. [Online]. Available: http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf.

[11] H. Wickham, "Tidy data," *The Journal of Statistical Software*, Submitted 2014.

[12] M. Dowle, T. Short, S. Lianoglou, A. S. with contributions from R. Saporta, and E. Antonyan, *data.table: extension of data.frame*, R package version 1.9.2, 2014. [Online]. Available: http://CRAN.R-project.org/package=data.table.

[13] K. Hamada, H. Mori, H. Shinoda, and T. M. Rutkowski, "Airborne ultrasonic tactile display brain-computer interface paradigm," in *Proceedings of the 6th International Brain-Computer Interface Conference*, accepted; in press, Graz, Austria, Apr. 2014. arXiv: 1404.4184 [q-bio.NC].

[14] R Core Team, *R: a language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/.

[15] M. Ahdesmaki, V. Zuber, S. Gibb, and K. Strimmer, *sda: shrinkage discriminant analysis and cat score variable selection*, R package version 1.3.3, 2014. [Online]. Available: http://CRAN.R-project.org/package=sda.

[16] T. Helleputte, *LiblineaR: linear predictive models based on the liblinear C/C++ library*, R package version 1.80-7, 2013.

[17] A. Delorme, *EEGLAB and supercomputing applications using free alternatives to MATLAB*, Aug. 2009. [Online]. Available: http://sccn.ucsd.edu/wiki/EEGLAB_and_supercomputing_applications_using_free_alternatives_to_Matlab.

[18] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[19] Revolution Analytics and S. Weston, *Foreach: foreach looping construct for r*, R package version 1.4.1, 2013. [Online]. Available: http://CRAN.R-project.org/package=foreach.

[20] R. Analytics, *doMC: foreach parallel adaptor for the multicore package*, R package version 1.3.3, 2014. [Online]. Available: http://CRAN.R-project.org/package=doMC.

[21] M. Heath, *Hypercube multiprocessors, 1986 : proceedings of the First Conference on Hypercube Multiprocessors, Knoxville, Tennessee, August 24-27, 1985*. Philadelphia: SIAM, 1986, ISBN: 0898712092.

[22] D. Eddelbuettel, *CRAN task view: high performance computing in R*, May 2014. [Online]. Available: http://cran.r-project.org/web/views/HighPerformanceComputing.html.

[23] *Does MATLAB pass parameters using "call by value" or "call by reference"?* MathWorks, Inc., Oct. 2012. [Online]. Available: http://www.mathworks.com/matlabcentral/answers/96960-does-matlab-pass-parameters-using-call-by-value-or-call-by-reference.

[24] K. Ushey, *RcppRoll: fast rolling functions through Rcpp and RcppArmadillo*, R package version 0.1.0, 2013. [Online]. Available: http://CRAN.R-project.org/package=RcppRoll.

[25] D. Eddelbuettel, *Seamless R and C++ Integration with Rcpp*. New York: Springer, 2013, ISBN 978-1-4614-6867-7.

[26] O. Mersmann, *Microbenchmark: sub microsecond accurate timing functions.* R package version 1.3-0, 2013. [Online]. Available: http://CRAN.R-project.org/package=microbenchmark.

[27] H. Wickham, "The split-apply-combine strategy for data analysis," *Journal of Statistical Software*, vol. 40, no. 1, pp. 1–29, 2011. [Online]. Available: http://www.jstatsoft.org/v40/i01/.