PARALLELISM STRATEGIES FOR NEUROPHYSIOLOGICAL DELAYED TRANSFER ENTROPY DATA PROCESSING: TOWARDS CAUSAL INFERENCE IN BIG DATA

JONAS ROSSI DOURADO[*], MICHEL BESSANI[*], DANIEL RODRIGUES DE LIMA[*], JOSÉ ROBERTO B. DE A. MONTEIRO[*], RAFAEL RODRIGUES MENDES RIBEIRO[*], CARLOS DIAS MACIEL[*]

[*]*Department of Electrical and Computational Engineering,*
University of São Paulo,
São Carlos, São Paulo, Brazil

Emails: jonas.dourado@usp.br, michel.bessani@gmail.com, drodriguesdelima@gmail.com, jrm@sc.usp.br, rafael.mendes.ribeiro@usp.br, maciel@sc.usp.br

Abstract— Nowadays, the amount of data being generated and collected has been rising with the popularization of technologies such as Internet of Things, social media, and smartphone. The increasing amount of data led the creation of the term big data. One class of Big Data hidden information is causality. Among the tools to infer causal relationships there is Delayed Transfer Entropy (DTE); however, it has a high demanding processing power. Many approaches were proposed to overcome DTE performance issues such as GPU and FPGA implementations. Our approach is to compare different parallel strategies to calculate DTE from neurophysiological time series using a heterogeneous Beowulf cluster aiming to increase DTE performance.

Keywords— Delayed Transfer Entropy. Neurophysiological Data. Causality. Parallelism Strategies. Big Data Analysis. Task Parallelism. Data Parallelism, Data Analysis Performance.

1 Introduction

Nowadays, the amount of data being generated and collected has been rising with the popularization of technologies such as Internet of Things, social media, and smartphone (Hashem et al., 2015). The increasing amount of data led the creation of the term big data, with one definition given by (Hashem et al., 2015), as a set of technologies and techniques to discover hidden information from diverse, complex and massive scale datasets. One class of hidden information is causality, which (Bareinboim and Pearl, 2016) discuss and propose a framework to deal with common found big data biases such as confounding and sampling selection.

Among the tools to infer causal relationships there are Mutual Information used by (Endo et al., 2015) to infer neuron connectivity and Granger causality used by (Strohsal et al., 2015) to model causality between US and UK economies. Additionally, exist Transfer Entropy (TE), which allows identification of cause-effect relationship by not accounting for simple and uniquely shared information (Schreiber, 2000). TE has been applied to many problems from diverse research fields e.g. finance (Yook et al., 2016); biosignals (Marzbanrad et al., 2015), complex networks (Haruna and Fujiki, 2016) and climatology (Hirata et al., 2016).

A derivation of TE metric called Delayed TE (DTE) is useful for neurophysiological causality as used by (Ito et al., 2011) to identify active connections between neurons and by (Wollstadt et al., 2014) to calculate information transfer and delays from magnetoencephalography signals. Despite TE and DTE wide applicability, they have a high demanding processing power (Shao et al., 2015), which is aggravated with large datasets as those found in big data. Many approaches were proposed to overcome performance issues such as an implementation using a GPU made by (Wollstadt et al., 2014) and an implementation using an FPGA made by (Shao et al., 2015). Another approach to speedup data analysis is using a computer cluster.

Parallel programs should be optimized to extract maximum performance from hardware on architecture case by case, which is far from trivial according to (Booth et al., 2016). There exist different and combined manners to explore parallelism such as data parallelism and task parallelism (Gordon et al., 2006). (Choudhury et al., 2015) stated that choosing the configuration of parallel programs is a "mysterious art" in a study which they created a model aiming maximum speedup by balancing different parallelism strategies for both cluster and cloud computer environments.

In this study, we compare parallel strategies to calculate DTE from neurophysiological time series using a heterogeneous Beowulf cluster aiming to increase DTE performance. We also analyze computing node performance within task parallelism to gain some insights to enrich parallel strategies discussion.

This paper is organized as follows: Until introduction end, it will be presented concepts that might help readers keep up with whole paper. In materials and methods, will be described all steps needed to reproduce the results. Results and discussion are self-explanatory. In conclusion, additionally, is suggested future works. Remaining information useful to reproducibility is located in an appendix to avoid nonessential noise through the text.

1.1 Beowulf Cluster

A Beowulf cluster is made by connecting consumer grade computers on a local network using Ethernet or other suitable connection technology (Yao et al., 2009). The term Beowulf cluster was coined by (Sterling et al., 1995), which created the topology on

NASA facilities as an alternative to expensive commercial vendor built High-Performance Clusters.

Beowulf cluster is widely used by diverse research fields such as Monte Carlo simulations(Yamakov, 2016), drug design (Moretti and Sartori, 2016), geographic data processing (Qin et al., 2014) and Electroencephalogram data processing (Yao et al., 2009).

1.2 Parallelism Strategies

According to (Booth et al., 2016), archiving parallel performance on chosen hardware architecture depends on factors such as scheduler overhead, data/task granularity, cache fitting and data synchronization. To do so, one can change the parallelism strategy to optimize performance.

There exist different abstraction level of parallelism strategies that can be combined (Choudhury et al., 2015). In this paper, two parallelism levels are independently explored, from the lower to higher: data level parallelism and task level parallelism.

Data parallelism strategy idea as stated by (Gordon et al., 2006), is when one processing data slice does not have dependency with next one. Thus, data is divided into several data slices and processing them equally by different processors.

Task parallelism objective is to spawn tasks across processors to speedup one scalable algorithm. Tasks can be spawned by a central task system or by a distributed task system, both adding processing overhead, with distributed task system achieving less overhead (Booth et al., 2016).

Often, a systematic comparison between parallelism strategies is necessary to verify which one has better performance (Booth et al., 2016).

1.3 IPython

IPython was born as an interactive system for scientific computing (Pérez and Granger, 2007), later receiving several improvements as parallel processing capabilities (IPython developers, 2011). These parallel processing capabilities become an independent package under IPython project and were renamed as ipyparallel (IPython developers, 2016a).

With minor code modification, ipyparallel enables a Python processing script to be distributed across a cluster, with minor script modifications (IPython developers, 2016b). Throughout the text, ipyparallel is referenced as IPython, since its documentation also refers to itself as IPython.

IPython already had been used in studies similar to our propose, as (Kershaw et al., 2015) used it for big data analysis in a cloud environment and (Stevens et al., 2013) authors were able to develop automated and reproducible neuron simulation analysis.

1.4 Surrogate

The word *surrogate* stands for something that is used instead of something else. In the case of surrogate signals (Dolan and Spano, 2001), the synthetic data used is randomly generated, but it also presents some characteristics of the original signal that it is taking place.

A surrogate signal has the same power spectrum that the original signal, but these two signals are uncorrelated. Different computational packages present several algorithms to generate surrogate signals (Magri et al., 2009), (Lindner et al., 2011). Surrogate data, represents, as best as possible, all the characteristics of the real process, though without causal interactions.

In the case of neurophysiological data, the causal association happens in phase synchronization (Yang et al., 2013). (Endo et al., 2015) used a surrogate data with the IAAFT (Iterative Amplitude Adjusted Fourier Transform) algorithm (Schreiber and Schmitz, 1996), which generates signals preserving the power density spectrum and probability density functions, but with the phase components randomly shuffled (Venema et al., 2006).

1.5 Transfer Entropy

Transfer Entropy (TE) measurement shown in Equation 1 was introduced by (Schreiber, 2000) and is useful to measure information transfer between two time series. TE has an asymmetric nature, being possible to determine information direction.

$$TE_{X\to Y} = \sum_{y_{n+1},y_n,x_n} p(y_{n+1},y_n^{(k)},x_n^{(l)}) \log_2 \frac{p(y_{n+1}|y_n^{(k)},x_n^{(l)})}{p(y_{n+1}|y_n^{(k)})} \quad (1)$$

where y_n and x_n denotes value of X and Y at time n; y_{n+1} the value of Y at time $n+1$; p is the probability of parenthesis content; l and k are the number of time slices used to calculate probability density function (PDF) using past values of X and Y, respectively; chosen log_2 means that TE results are given in bits.

Assuming $k = 1$ and $l = 1$ to simplify analysis (Also called as D1TE by (Ito et al., 2011)), TE algorithm is demanding regarding computational power (Shao et al., 2015), with its computational complexity being $O(B^3)$, where B is the chosen number of bins in PDF.

An extension to D1TE proposed by (Ito et al., 2011) is delayed transfer entropy (DTE - Equation 2), which is a D1TE with variable causal delay range. This way, a parameter d represents a variable delay between y and x. DTE is a useful metric to determine where within d range, occurs the biggest transfer of information from X to Y.

$$TE_{X\to Y}(d) = \sum_{y_{n+1},y_n,x_{n+1-d}} p(y_{n+1},y_n,x_{n+1-d}) \log_2 \frac{p(y_{n+1}|y_n,x_{n+1-d})}{p(y_{n+1}|y_n)} \quad (2)$$

2 Material and Methods

This study emerged from recurrent cluster usage in our lab, demanded by several applications as multi-scenarios Monte Carlo simulations (Bessani et al., 2016), optimization of large-scale systems reconfiguration (Camillo et al., 2016), and in specific biosignals analysis with DTE (de Lima et al., 2016). Studies with DTE usage were negatively affected by high processing power demands (Shao et al., 2015), therefore often limiting data size scope or even number of surrogate datasets for DTE analysis.

With an objective to clarify program flow, the serial version is shown in Algorithm 2. The program calculates Embedding parameters for each channel and calculate DTE for each two channel permutation. Last, surrogate signals representing each permutation are generated, and DTEs are calculated. The most surrogate, the better, since it contributes to increasing causality statistical evidence.

Embedding was calculated in order to find the target variable's past, which can be found in the first local minimum of auto correlation measures (Kantz and Schreiber, 2004).

Algorithm 1 Execute DTE with surrogate

```
1:  for experiment in experiment_list_file do
2:      experiment_signal ← load_signal_from_disk (experiment)
3:      for each channel in experiment_signal do
4:          calculate embedding(channel)
5:      end for
6:      for each two-channel permutation in experiment_signal do
7:          calculate DTE(channel1, channel2, channel2_embedding)
8:      end for
9:      for i = 0 to num_surrogates do
10:         surrogate ← generate_surrogate (experiment_signal)
11:         for each two-channel permutation in surrogate do
12:             calculate DTE(channel1, channel2, channel2_embedding)
13:         end for
14:     end for
15: end for
```

The previous existent code comes from several years of development, is arbitrarily named as CODE A and is shown as a flow chart in Figure 1. CODE A is a Python (van Rossum and Drake, 2011) script which uses libraries such as numpy (Walt et al., 2011), IPython (Pérez and Granger, 2007) and lpslib (developed in-house by LPS laboratory). It tackled performance issues by using data parallelism, made possible by IPython *ipyparallel* library.

On CODE A, for each experiment, signals are read on host node, Embedding code is executed on host node, Embedding pushes corresponding channel data to all computing nodes, each computing node process one part of data and results are gathered on the host node. Note that Embedding is executed once for each channel. After Embedding, DTE is executed on host node for each two channel permutation, it pushes channel data to all computing nodes, each node process one part of data and result is gathered on the host node.

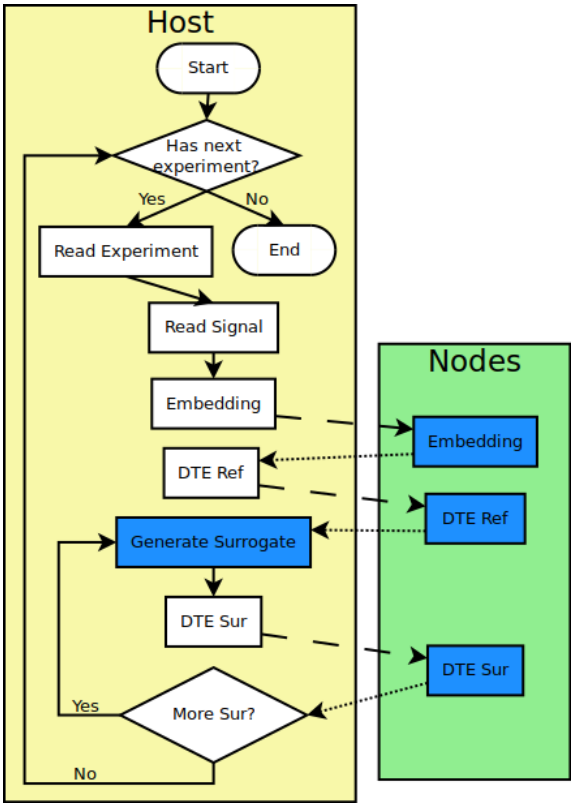

Figure 1: CODE A- Data parallelism program. Note that each dashed line means data transfer to every cluster node and every dotted line means synchronization to host program flow.

The decision to try another parallelism strategy come from analyzing CPU load during CODE A execution and observing a processor sub-utilization. This observation was done by executing *htop* (Bartosz Fenski et al., 2016) program on a computing node and checking that system load average was significantly smaller than the number of processors. By reading Linux *proc* manual (Linux Developers, 2016), DTE calculation clearly wasn't in run queue or waiting for disk I/O to fully load (meaning the load average is equal the number of processors) each node.

Further investigation by adding extensive logging facilities to CODE A confirmed that the low load average was caused by network communication bottle-

neck.

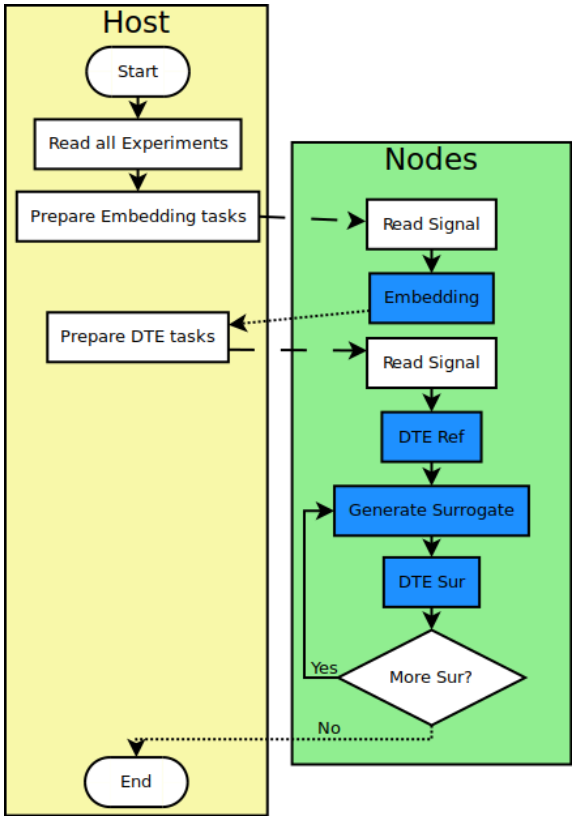

Figure 2: CODE B - Task parallelism program. Note that each dashed line means task parameters transfer to individual cluster node and every dotted line means synchronization to host program flow.

Aiming to mitigate network communication overhead, code were refactored to use task-based parallelism. The idea is to load signal data from main storage instead network transfer from host node. Refactored code were named as CODE B and is shown in Figure 2.

Before executing CODE B, every signal data must be copied to each computing node. The main difference from CODE A is that every experiment is read once, every Embedding tasks are executed, and finally, one task is created for each two channel permutation for every experiment. Tasks are asynchronously executed and scheduled by IPython.

Both CODE A and CODE B were executed with a different number of surrogates (1, 5, 10 and 20) to compare performance between them, except 20 surrogates for CODE A due excessively long execution time (estimated in more than 6000 minutes by extrapolating results from CODE A with a smaller number of surrogates). The long execution time of each experiment made nonviable repeating it to calculate standard deviation, but it was checked for 1 surrogate case that time variance between executions was minimal.

The system used to analyze performance difference was a heterogeneous Beowulf cluster composed of 10 nodes connected through Gigabit Ethernet Switch model HP Procurve 1910-24G. Nodes hardware configuration is listed in Table 2 and software configuration is listed in Table 3 and are presented in an appendix.

Logs were processed to calculate duration for each execution. Linear least square method was used to fit a line for data and task parallelism duration. Supposing surrogate creation duration is insignificantly in comparison with DTE duration; each line slope represents Minutes/surrogate.

With line slopes, speedup was calculated to measure performance gain from task parallelism over data parallelism.

3 Results and Discussion

Analyzing logs gave results shown by point marks and fitted line using linear least square method in Figure 3.

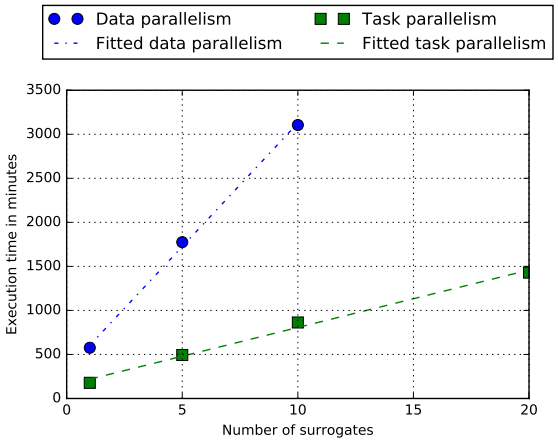

Figure 3: Data and task parallelism execution time versus number of surrogate signals. Point marks shows numerical results. Lines show data fitted by linear square method

Lines slope for each parallelism strategy are 280.385 minutes/surrogate (data parallelism) and 65.257 minutes/surrogate (task parallelism). Therefore, speedup can be determined in Equation 3.

$$Speedup = \frac{t_{data\ parallelism}}{t_{task\ parallelism}} = \frac{280.385}{65.257} = 4.297 \quad (3)$$

Achieved speedup ~ 4.3 shows that task parallelism is significantly faster than data parallelism. After analyzing logs, positive speedup can be explained by three main factors and negatively impacted by another.

First speedup explanation is data locality since data is stored on local disk in task parallelism versus being transferred by the network in data parallelism. Also, former has to transfer channel data for every surrogate, while later, locally read signal data only once for each two channel permutation surrogates.

Second factor is sub-optimum node utilization caused by cluster heterogeneity illustrated in Figure 4. This happens in data parallelism because data

is equally divided across computing nodes with different performance, causing faster nodes, which finished data processing, to wait for slower nodes.

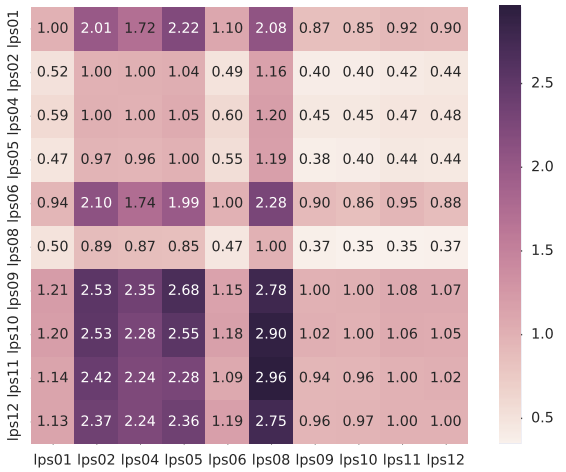

Figure 4: DTE task performance comparison between different computing nodes to highlight cluster heterogeneity. In task parallelism, each experiment spawned $number\ of\ channels * (number\ of\ channels - 1)$ DTE tasks of the same size across computing nodes, their execution times were used to calculate speedup of Y axis computing node over X axis computing node and finally was made an average of calculated speedups for each cell. Execution log used to generate this plot was from 20 surrogates. Values are given in relative speedup

The third factor is caused by the fact of data parallelism surrogate datasets are generated only by host node, forcing all computing nodes to wait for surrogate dataset generation. Task parallelism does not suffer from the same problem, as while one surrogate dataset is generated by one computing nodes, it does not block another computing node.

Negatively affecting task parallelism speedup is caused by asynchronous nature of tasks, when task pool is exhausted, some computing nodes are left without any task.

4 Conclusion

Using task parallelism strategy to increase DTE algorithm performance in a heterogeneous cluster was shown as a faster alternative in comparison with data parallelism. Having in sight big data trend, it is a significant result, since it will enable causal inference for bigger datasets or with better causality statistical evidence.

Having verified task parallelism as a better approach to DTE in a Beowulf cluster, it remains open how the number of computing nodes affects performance. Thus, future research should investigate how scalable task parallelism is after increased number of computing nodes, with a nice addition of using smaller artificial data feasible to execute multiple times each experiment. Another further investigation might also check if the amount of RAM has a correlation with performance in computing nodes when executing task parallelism. Moreover, different parallelism strategies can be tested on a case by case aiming to speedup processing of the ever increasing data size.

Acknowledgments

The authors would like to thank **CAPES** – Brazilian Federal Agency for Support and Evaluation of Graduate Education within the Ministry of Education of Brazil, **CNPq** – National Council of Technological and Scientific Development – Project 475064/2013-5 for supporting this work, **FAPESP** – São Paulo Research Foundation for supporting this work and previous people who worked on program source code: Carlos Dias Maciel, Wagner Endo, Fernando Pasquini Santos and Daniel Rodrigues de Lima.

We also would like to thank Sinal Processing Laboratory (**LPS**) and the University of São Paulo (**USP**) for providing the infrastructure, in special the Beowulf cluster used in this study.

References

BAKER, M. (2016). Is there a reproducibility crisis?, *Nature* **533**: 452–454.

Bareinboim, E. and Pearl, J. (2016). Causal inference and the data-fusion problem, *Proceedings of the National Academy of Sciences* **113**(27): 7345–7352.

Bartosz Fenski et al. (2016). *htop(1) Linux User's Manual*.

Bessani, M., Fanucchi, R. Z., Delbem, A. C. C. and Maciel, C. D. (2016). Impact of operators' performance in the reliability of cyber-physical power distribution systems, *IET Generation, Transmission & Distribution* **10**: 2640–1646.

Booth, J. D., Kim, K. and Rajamanickam, S. (2016). A comparison of high-level programming choices for incomplete sparse factorization across different architectures, *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, IEEE, pp. 397–406.

Camillo, M. H., Fanucchi, R. Z., Romero, M. E., de Lima, T. W., da Silva Soares, A., Delbem, A. C. B., Marques, L. T., Maciel, C. D. and London, J. B. A. (2016). Combining exhaustive search and multi-objective evolutionary algorithm for service restoration in large-scale distribution systems, *Electric Power Systems Research* **134**: 1–8.

Choudhury, O., Rajan, D., Hazekamp, N., Gesing, S., Thain, D. and Emrich, S. (2015). Balancing thread-level and task-level parallelism for data-intensive workloads on clusters and clouds, *2015*

IEEE International Conference on Cluster Computing, IEEE, pp. 390–393.

de Lima, D. R., Santos, F. P. and Maciel, C. D. (2016). Network structural reconstruction base on delayed transfer entropy and synthetic data., *CBA 2016*, pp. 1–6.

Dolan, K. T. and Spano, M. L. (2001). Surrogate for nonlinear time series analysis, *Physical Review E* **64**(4): 046128.

Editorial, N. (2016). Reality check on reproducibility, *Nature* **533**: 437.

Endo, W., Santos, F. P., Simpson, D., Maciel, C. D. and Newland, P. L. (2015). Delayed mutual information infers patterns of synaptic connectivity in a proprioceptive neural network, *Journal of Computational Neuroscience* **38**(2): 427–438.

Gordon, M. I., Thies, W. and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, *SIGARCH Comput. Archit. News* **34**(5): 151–162.

Haruna, T. and Fujiki, Y. (2016). Hodge decomposition of information flow on small-world networks, *Frontiers in Neural Circuits* **10**.

Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A. and Khan, S. U. (2015). The rise of "big data" on cloud computing: Review and open research issues, *Information Systems* **47**: 98–115.

Hirata, Y., Amigó, J. M., Matsuzaka, Y., Yokota, R., Mushiake, H. and Aihara, K. (2016). Detecting causality by combined use of multiple methods: Climate and brain examples, *PLoS ONE* **11**(7): 1–15.

IPython developers (2011). Ipython 3.2.1 documentation - 0.11 series.

IPython developers (2016a). ipyparallel 5.2.0 documentation - changes in ipython parallel.

IPython developers (2016b). ipyparallel 5.2.0 documentation - ipython parallel overview and getting started.

Ito, S., Hansen, M. E., Heiland, R., Lumsdaine, A., Litke, A. M. and Beggs, J. M. (2011). Extending transfer entropy improves identification of effective connectivity in a spiking cortical network model, *PLoS ONE* **6**(11).

Kantz, H. and Schreiber, T. (2004). *Nonlinear time series analysis*, Vol. 7, Cambridge university press.

Kershaw, P., Lawrence, B., Gomez-Dans, J. and Holt, J. (2015). Cloud hosting of the ipython notebook to provide collaborative research environments for big data analysis, *EGU General Assembly Conference Abstracts*, Vol. 17, p. 13090.

Lindner, M., Vicente, R., Priesemann, V. and Wibral, M. (2011). Trentool: A matlab open source toolbox to analyse information flow in time series data with transfer entropy, *BMC neuroscience* **12**(1): 1.

Linux Developers (2016). *proc(5) Linux User's Manual*.

Magri, C., Whittingstall, K., Singh, V., Logothetis, N. K. and Panzeri, S. (2009). A toolbox for the fast information analysis of multiple-site lfp, eeg and spike train recordings, *BMC neuroscience* **10**(1): 1.

Marzbanrad, F., Kimura, Y., Palaniswami, M. and Khandoker, A. H. (2015). Quantifying the interactions between maternal and fetal heart rates by transfer entropy, *PloS one* **10**(12): e0145672.

Moretti, L. and Sartori, L. (2016). A simple and resource-efficient setup for the computer-aided drug design laboratory, *Molecular Informatics* pp. n/a–n/a.

Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing, *Computing in Science and Engineering* **9**(3): 21–29.

Qin, C.-Z., Zhan, L.-J., Zhu, A.-X. and Zhou, C.-H. (2014). A strategy for raster-based geocomputation under different parallel computing platforms, *International Journal of Geographical Information Science* **28**(11): 2127–2144.

Schreiber, T. (2000). Measuring Information Transfer, *Physical Review Letters* **85**(2): 19.

Schreiber, T. and Schmitz, A. (1996). Improved surrogate data for nonlinearity tests, *Phys. Rev. Lett.* **77**: 635–638.

Shao, S., Guo, C., Luk, W. and Weston, S. (2015). Accelerating transfer entropy computation, *Proceedings of the 2014 International Conference on Field-Programmable Technology, FPT 2014* pp. 60–67.

Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A. and Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation, *In Proceedings of the 24th International Conference on Parallel Processing*, CRC Press, pp. 11–14.

Stevens, J.-L. R., Elver, M. and Bednar, J. A. (2013). An automated and reproducible workflow for running and analyzing neural simulations using lancet and ipython notebook, *Frontiers in neuroinformatics* **7**: 44.

Strohsal, T., Proaño, C. R., Wolters, J. et al. (2015). How do financial cycles interact? evidence from the us and the uk, *Technical report*, Sonderforschungsbereich 649, Humboldt University, Berlin, Germany.

van Rossum, G. and Drake, F. L. (2011). *The Python Language Reference Manual*, Network Theory Ltd.

Venema, V., Ament, F. and Simmer, C. (2006). A stochastic iterative amplitude adjusted fourier transform algorithm with improved accuracy, *Nonlinear Processes in Geophysics* **13**(3): 321–328.

Walt, S. v. d., Colbert, S. C. and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation, *Computing in Science and Engg.* **13**(2): 22–30.

Wollstadt, P., Martínez-Zarzuela, M., Vicente, R., Díaz-Pernas, F. J. and Wibral, M. (2014). Efficient transfer entropy analysis of non-stationary neural time series, *PLoS ONE* **9**(7): 1–21.

Yamakov, V. I. (2016). Parallel grand canonical monte carlo (paragrandmc) simulation code, *Technical report*, NASA.

Yang, C., Jeannès, R. L. B., Faucon, G. and Shu, H. (2013). Detecting information flow direction in multivariate linear and nonlinear models, *Signal Processing* **93**(1): 304–312.

Yao, Y., Chang, J. and Xia, K. (2009). A case of parallel eeg data processing upon a beowulf cluster, *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, IEEE, pp. 799–803.

Yook, S.-H., Chae, H., Kim, J. and Kim, Y. (2016). Finding modules and hierarchy in weighted financial network using transfer entropy, *Physica A: Statistical Mechanics and its Applications* **447**: 493–501.

Appendix

Observing recent concerns about experiment reproducibility across every science field (Editorial, 2016) (BAKER, 2016), this paper aims to achieve complete reproduction. Therefore in this appendix is shown additional information useful to reproduce it.

Source code

All code was managed using Git version control software within a private repository. Exact code revisions used by this study is shown in Table 1.

Cluster configuration

Cluster hardware configuration is listed is Table 2 and software configuration is listed in Table 3

Dataset

The dataset is composed of 35 neurophysiological signals each with four simultaneous captured channels. The average number of signal samples is about 1 million samples with standard deviation about 500 thousand samples.

Table 1: Code Git revisions hash

Parallel strategy	Revision hash
Data parallelism	f85aac7e8ff46c74b8e758211197dfc8b069571d
Task parallelism	e97a687c51cfad61ac097fb5fc26b029967615da

Table 2: Cluster hardware configuration. RAM modules were list separately since some nodes has multiple memory modules to explore dual channel. Main storage describe storage media used during script execution, some nodes might have other unused storage media.

Node	Processor (cores)	RAM (speed)	Main Storage size (model)	Ethernet
host	i5-2500 CPU @ 3.30GHz	4 + 4 GiB (1333MHz)	2TB WDC WD20EARX-00P	Gigabit
lps01	i7-4770 CPU @ 3.40GHz (8)	8 + 8 GiB (1333MHz)	1TB ST1000DM003-1CH1	Gigabit
lps02	i7-3770 CPU @ 3.40GHz (8)	8 GiB (1333MHz)	60GB KINGSTON SV300S3	Gigabit
lps04	i7-4820K CPU @ 3.70GHz (8)	8 GiB (1333MHz)	2TB ST2000DM001-1CH1	Gigabit
lps05	i7-4820K CPU @ 3.70GHz (8)	8 GiB (1333MHz)	1863GiB ST2000DM001-1CH1	Gigabit
lps06	i7-4820K CPU @ 3.70GHz (8)	8 + 8 GiB (1333MHz)	60GB KINGSTON SV300S3	Gigabit
lps08	i7 950 CPU @ 3.07GHz (8)	4 + 4 + 4 GiB (1066MHz)	2TB ST32000542AS	Gigabit
lps09	i7-4790 CPU @ 3.60GHz (8)	8 + 8 GiB (1600MHz)	256GB SMART SSD SZ9STE	Gigabit
lps10	i7-4790 CPU @ 3.60GHz (8)	8 + 8 GiB (1600MHz)	256GB SMART SSD SZ9STE	Gigabit
lps11	i7-4790 CPU @ 3.60GHz (8)	8 + 8 GiB (1600MHz)	256GB SMART SSD SZ9STE	Gigabit
lps12	i7-4790 CPU @ 3.60GHz (8)	8 + 8 GiB (1600MHz)	256GB SMART SSD SZ9STE	Gigabit

Table 3: Cluster software configuration. Updated at shows when each cluster node was last fully updated.

Node	Operating System (updated at)	numpy	IPython	pyfftw	Linux kernel
host	Fedora 24 Workstation (2016-08-17)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps01	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps02	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps04	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps05	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps06	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps08	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps09	Fedora 24 Workstation (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps10	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps11	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64
lps12	Fedora 24 Server (2016-08-16)	1.11.0	3.2.1	0.10.3.dev0+e827cb5	4.6.6-300.fc24.x86_64