An Efficient Implementation of the Synchronization Likelihood Algorithm for Functional Connectivity

Francisco Rosales · Antonio García-Dopico · Ricardo Bajo · Ángel Nevado

Abstract Measures of functional connectivity are commonly employed in neuroimaging research. Among the most popular measures is the Synchronization Likelihood which provides a non-linear estimate of the statistical dependencies between the activity time courses of different brain areas. One aspect which has limited a wider use of this algorithm is the fact that it is very computationally and memory demanding. In the present work we propose new implementations and parallelizations of the Synchronization Likelihood algorithm with significantly better performance both in time and in memory use. As a result both the amount of required computational time is reduced by 3 orders of magnitude and the amount of memory needed for calculations is reduced by 2 orders of magnitude. This allows performing analyses that were not feasible before from a computational standpoint.

Keywords Functional connectivity · Synchronization likelihood · Implementation · Parallelization · OpenMP · GPU

Introduction

Two notions coexist in brain functioning: segregation and integration. The functional segregation of information among distinct local areas (differing in anatomy and physiology) contrasts with its global integration during perception and behavior (Tononi et al. 1994).

In a recent review (Singer 2013), the brain is defined as a "complex, self-organised system, in which principles of distributed, parallel processing coexist with serial operations within highly interconnected networks".

Functional connectivity (FC) measures resulting from calculating the statistical dependencies among the activity time-course of different brain regions (Friston 1994) are typically obtained from neuroimaging recordings to provide an index of how brain regions are coordinated to support higher cognitive functions. Long range synchronization between signals originated in relatively distant neuronal populations have been proposed as the mechanism for communicating and integrating the information in the brain (Varela et al. 2001; Fries 2005). It should be noted that FC is a different concept to both anatomical and effective connectivity (Friston 1994; Niso et al. 2013). Anatomical connectivity refers to the existence of tracts or fibres linking the brain areas. The term effective connectivity is used to convey that one area is directly influencing the other.

FC is considered to be an essential "tool" for the study of both healthy and pathological brain function (Singer 1999; Varela et al. 2001; Buzsáki and Draguhn 2004). There is increasing evidence of differences in FC between patients and control groups in a large number of condition such as schizophrenia, autism and Alzheimer disease (Singer 2013; Guggisberg et al. 2008; Stam et al. 2009; Bajo et al. 2010; Castellanos et al. 2010).

Many FC measures have their roots on dynamical systems or information theory and can be classified as reflecting either generalized or phase synchronization (Pereda et al. 2005). Among the generalized synchronization indexes, the Synchronization Likelihood (SL) (Stam and Van Dijk 2002) is arguably the most popular measure for neurophysiological data (Pijnenburg et al. 2004; Ahmadlou et al. 2012; Buldú et al. 2011; Montez et al. 2006).

The SL provides a normalized estimate of the dynamical interdependencies between two simultaneously recorded time series. It is closely related to the concept of generalized mutual information (Buzug et al. 1994) and relies on the detection of simultaneously occurring patterns in the two time series, which can be complex and widely different across signals. Besides, it is robust and sensitive to nonlinear dependencies.

Two important limitations for the practical use of the SL are its computational and memory costs of current implementations. Although references in the literature to the computational demands have been softening over time, ("computationally prohibitive" (Stam et al. 2003), "very computationally demanding" (David et al. 2004), "computationally demanding" (Montez et al. 2006) or "not computationally efficient" (Acharya et al. 2010)) probably alongside the progress in computational capabilities of computers, the limitation is still serious.

In the present work we present three new very efficient implementations of the SL algorithm: a sequential version and two parallelizations based on the OpenMP and CUDA architectures.

Methods

The Synchronization Likelihood Algorithm

The Synchronization Likelihood (SL) measures the generalized synchronization between two signals. It is sensitive to whether when one signal repeats its pattern of activity other signals tend to repeat their own pattern at the same times.

The SL algorithm uses the observation time-series of dynamical systems in order to measure the degree of synchronization or coupling between each pair of signals. When applied to more than two signals, all calculations should be done simultaneously in order to reduce the number of redundant computations.

Phase one: Construction of the Time-delay Embedding Vectors

Typically we do not have access to all the variables that characterize a dynamical system but rather to a subset or combination of them. The behaviour of an underlying system can be characterized from a time series of observations. The state of the system at any time instant can be represented by a time-delay embedding vector (Takens 1981), which is a set of values of the signal in a given time-window. Takens proved that, if the time-delay vector is chosen appropriately, the reconstructed vectors convey the fundamental characteristics of the underlying system such as degrees of freedom, dependence on initial conditions and dynamics, and that recurrent states of the system are represented by similar time-delay vectors. (Takens 1981; Posthuma et al. 2005)

For each single observation of a signal x_i (where i denotes the time instant), a state vector X_i is defined as,

$$X_i = (x_i, x_{i+L}, x_{i+2L}, ..., x_{i+(m-1)L}) \qquad (1)$$

where m is the embedding dimension and L is the time lag between chosen samples. The optimal values of these parameters are a function of the frequency band of interest (Montez et al. 2006). The lag should be chosen so that the highest frequency is sampled at least twice per cycle, and the embedding dimension should be such that at least one whole cycle of the slowest oscillations is captured (Betzel et al. 2012).

Of course, there is a gap of $(m-1)L$ samples at the end of the time-series where X_i cannot be defined with the same dimension. This is a boundary situation to take into account.

Phase two: Localization of the Recurrent Dynamical States

The probability $P^{\varepsilon}_{X,i}$ that state vectors X_j in the time-interval defined by parameters $w1$ and $w2$ are closer than certain distance ε to a reference state vector X_i at time instant i can be defined (Stam and Van Dijk 2002) as

$$P^{\varepsilon}_{X,i} = \frac{1}{2\times(w2-w1)} \sum_{j\in[\pm w2]\notin[\pm w1]} \theta\left(\varepsilon - |X_i - X_{i+j}|\right) \qquad (2)$$

The distance || between state vectors can be the Euclidean distance or any other such the maximum norm. The Heaviside step function $\theta(d)$ is 1 when its argument d is positive and 0 otherwise.

The $w1$ parameter establishes an exclusion window around the time instant i where similar state vectors are not likely to represent a recurrence but rather that the system has not had time to evolve (Theiler 1986).

The $w2$ parameter establishes an inclusion window around the time instant i that sharpens the time resolution of the synchronization measure (Stam and Van Dijk 2002).

It has to be large enough for P_i^{ε} to make sense as the proportion of vectors considered as recurrences (Montez et al. 2006).

The combination of $w1$ and $w2$ establishes the surroundings of i given by the subintervals $[i - w2, i - w1)$ and $(i + w1, i + w2]$.

The original description of Stam and Van Dijk (2002) states that:

"Now for each X and each i the critical distance $\varepsilon_{X,i}$ is determined for which $P_{X,i}^{\varepsilon_{X,i}} = p_{ref}$, where $p_{ref} \ll 1$"

A literal interpretation of this statement would lead to a naive algorithm implementation to find changing values of the "critical distance $\varepsilon_{X,i}$" for each signal X and each time instant i, but this is far from necessary, because p_{ref} is not an unknown but one of the SL algorithm input parameters.

When applied to Eq. 2, the p_{ref} parameter represents the fraction of state vectors inside the time subintervals $[i - w2, i - w1)$ and $(i + w1, i + w2]$, which are to be considered closer to Xi than the critical distance. So, as p_{ref}, $w1$ and $w2$ are constant values for any signal and for any time instant[1], then the number of state vectors closer than the distance is also a constant integer, and equal to: $N = \lceil 2 \times (w2 - w1) \times p_{ref} \rceil$.

The roundup operator $\lceil\rceil$ is used to obtain a non zero integer. So the actual effective p_{ref} which is used is: $Effective\ p_{ref} = N/(2 \times (w2 - w1))$.

What we need to determine therefore, for any signal X and time instant i, is the set $C_{X,i}$ of time instants j relative to i where the Heaviside step function θ takes value 1. In other words, to identify where, within the defined interval, the N most similar state vectors to the reference state vector X_i are. As we will see, this set of points is all that is necessary for the next phase of the algorithm.

Phase three: Likelihood of Simultaneous Recurrent States in two Signals

The SL measures the likelihood that there is recurrence in the simultaneous states of the two systems.

The number of simultaneous repetitions for signals X and Y around time instant i is defined as:

$$n_{XY,i} = \sum_{j \in [\pm w2] \notin [\pm w1]} \theta\left(\varepsilon_{X,i} - |X_i - X_{i+j}|\right) \theta\left(\varepsilon_{Y,i} - |Y_i - Y_{i+j}|\right) \tag{3}$$

It follows that the Synchronization Likelihood at time instant i is just the number recurrences over the total possible number of recurrences in the time interval considered:

$$SL_i = \frac{n_{XY,i}}{N} \tag{4}$$

[1]The problem of dealing with the head and the tail of the time series will be discussed later.

Phase four: Computation of the SL Across a Time Interval

Usually, the SL is expressed as an average across time of SL_i. can be reduced by computing SL_i at time steps larger than the sampling interval of the original signal at the cost of losing precision and time resolution. The original algorithm implementation calls this step parameter *speed*[2](Calmels et al. 2008). Caution should be exerted when using this speeded-up calculation as SL_i can change rapidly on time scales smaller than $w1$ (Montez et al. 2006).

Low Level Description of the Implementation

Although we had access to a MATLAB[3] implementation of the algorithm from the original author's group termed "Synchronization Likelihood Stand-Alone Calculator", we decided to use it only as a reference. Instead we chose to develop our own complete reimplementation of the algorithm in the C language, starting from the analysis of the algorithm as shown in the previous section. The C language is very suitable for optimization because of the level of control it allows. The compiler can be tuned to produce efficient code in terms of computational time and memory requirements. Additionally there is a large set of tools and technologies available to produce all kinds of parallel code.

Interfaces and Applications

The pseudocode of the algorithm implementation, FSL_delta, is provided in Fig. 1. Two different versions are used to compute SL_i for all the signals at time i. The "generalized" version is used at the beginning and end of the time series and takes into account boundary effects. It works by narrowing the considered surrounding intervals, changing $w1$, $w2$ and m as i approaches the edges of the time series. This method is completely general in the sense that it is able to cope with any i including those close to the end of the time series. This version is particularly useful for short datasets. A second "specialized" version is suitable only for the middle part of the time series, where boundary effects are not an issue. This is a specialized more efficient method. Both methods have the same external interface and the same internal structure.

The external interface allows to serially process any amount of data in finite increments of any size, and for any number of synchronized signals. This allows the time i to monotonously grow at any rate, and with any constant *speed* value. Therefore it can be applied to the processing of

[2]The BrainWave and the MATLAB versions also use the speed parameter

[3]http://www.mathworks.com

```
FSL_delta(L,m,w1,w2,pref,speed,data[I][K],Ib,Ie,SL[K][K])
{
  /* Size of the set of closest vectors */
  $N = \lceil 2 \times (w2 - w1) \times p_{ref} \rceil$

  /* Main time loop across data */
  For each time instant $i \in [Ib...Ie]$ at step $speed$

    /* First subloop */
    For each channel $X \in K$
      /* Localize the recurrent dynamical states */
      For each $j \in [\pm w2] \notin [\pm w1]$
        Calculate $|X_i - X_{i+j}|$
        /* On-the-fly time-embedding vectors access */
        Remember the set of the $N$ closest $j$'s

    /* Second subloop */
    For each channel $X \in K$
      For each channel $Y \in K$
        /* Likelihood of simultaneous recurrent state */
        $n_{XY,i}$ = count of simultaneous $j$'s
                     in both closest sets
        $SL[X][Y] += n_{XY,i}/N$

  /* To externally average SL along the time */
  Return the number of time steps carried out
}
```

Fig. 1 Pseudocode of the FSL_delta method

pre-loaded data as well as to the processing of streams of data, as in the case of real-time applications.

Internal Structure and Features

As Fig. 1 shows, to optimize computational time, the internal structure of the FSL_delta method does not literally follow the phases described previously. For example, the "construction of the time-delay embedding vectors" phase is done implicitly. The raw data are accessed directly in the right order as the state vectors components are needed. Obviously, this eliminates the need to store the vectors in memory. The "localization of the recurrent dynamical states" phase can be done independently for each signal, and the way it is done critically affects the computational efficiency. To determine which are the "N" shortest distances between the reference state vector and each one of the state vectors in its surrounding intervals, all distances need to be calculated. Only the N shortest distances need to be stored which can be done with insertion sorting. There is no need to determine ε. All that needs to be temporarily stored is the j relative position of the N closest vectors to i.

Optimization techniques try to avoid the repetition of prior calculations. Values that may be needed later should ideally be kept in memory for later reuse. Although we have evaluated this through several alternatives, for the present algorithm the results indicate that this does not constitute an advantage given the computing power of current CPUs and current compiler performance. The computation of the "likelihood of simultaneous recurrent state between each pair of signals" phase should be done after all the corresponding sets of closest points have been computed for all the signals. Each set contains only N relative j positions of the closest vectors, and by temporarily transforming it into an indexable bytemap, the cross comparison between signals is trivial and very fast.

As suggested by Stam et al. (2003), the Synchronization Entropy (Hs), which gives the spatio-temporal variability of synchronization, can be computed from the Synchronization Likelihood (SL) at a negligible additional cost.

Finally, the "computation of SL across a time-interval" is not done by the FSL_delta method. The number of coincidences is accumulated in global data structures. The averaging along the desired period of time is done externally and can be done at any moment. In this way, we can compute, not only the global SL, but also a partial SL for any smaller time interval as defined by parameter T.

It should be noticed that the computations for different values of i inside the FSL_delta method are mostly independent. This allows for parallelization, as we will see below.

Description of the Different SL Implementations

We compare two different SL algorithm implementations, the Fast SL implementation (FSL) described in this article, and the original SL algorithm (Orig) as described in Stam and Van Dijk (2002). The most similar version to the original one we have had access to is a MATLAB script called `sync_sa.m`, entitled "Synchronization Likelihood Stand-Alone Calculator" and dated July 2003. As it is a bit outdated, we also consider *BrainWave*,[4] the Java based application for functional connectivity and network analysis currently supported by the group of C.J. Stam. Whenever possible we also consider a parallelization of the different implementations.[5]

The different algorithms (Table 1 have been named after the algorithm implementation (Orig or FSL), the computer language used (MATLAB, Java or C) and the type of parallelization applied and are the following:

Orig-Matl-MulTh: This is the `sync_sa.m` MATLAB script executed with the options "-nodesktop" and "-nosplash" to reduce the memory footprint. Because MATLAB uses the Intel Math Kernel Library internally, which includes a multithreaded version of BLAS (Basic Linear Algebra Subroutines), the elementary library functions with vector arguments are multithreaded. Therefore this constitutes a multithread execution.

Table 1 The different SL implementations evaluated in this article

Version name	Algorithm implementation	Language	Parallelization
Orig-Matl-Seq	The original	MATLAB	Sequential
Orig-Matl-MulTh	The original	MATLAB	Multithreaded
Orig-Matl-parfor	The original	MATLAB	Multiple CPU cores
Orig-Java-Seq	Supposedly the original	Java	Sequential
Orig-C-Seq	The original	C	Sequential
FSL-C-Seq	Fast SL	C	Sequential
FSL-C-OMP	Fast SL	C	Multiple CPU cores
FSL-C-CUDA	Fast SL	C	Graphic processing units

Orig-Matl-Seq: This adds the option "-singleCompThread" to the previous version to limit MATLAB (and the underlying MKL library) to a single computational thread. This version is consequently executed sequentially.

Orig-Matl-parfor: This is an explicitly parallel version of the `sync_sa.m` script with only minimal changes. The MATLAB `parfor` sentence is used for the main parallel loop and a matlabpool is opened to use multiple processors. The number of processors is limited to 8 by the toolbox.

Orig-Java-Seq: This is the *BrainWave*'s SL algorithm implementation. The source code is not available, but the application can be downloaded for free in a Java precompiled form. We assume that this implementation of the SL remains similar to the original one. No parallelization seems to have been implemented.

Orig-C-Seq: This is a C translation of the original algorithm, resulting from rewriting the `sync_sa.m` script in the C language without modifying the algorithm. It is used for comparing the MATLAB interpreted language efficiency with that of a fully compilable language such as C. All the MATLAB matricial operations were written directly in C as nested loops, without using additional libraries. Any possible code vectorization is automaticaly done by the compiler.

FSL-C-Seq: This is a fully optimized sequential C code implementation of the Fast SL suitable for compiler vectorization as described before.

FSL-C-OMP: This is an adaptation of the sequential version using the OpenMP standard[6] (Dagum and Menon 1998) to take advantage of current shared memory computer architectures to execute the code in parallel over multiple local CPU cores. In a straightforward adaptation, only the main time loop of the FSL_delta method has been parallelized using OpenMP directives. Therefore, different time instants will be executed in parallel by different threads over the available CPU cores, but both subloops will be executed one after the other by the same thread. The first subloop has no dependencies between channels. For the second subloop, the partial result obtained by each thread must be accumulated, so the access to the variable that contains the final results must be synchronized using an atomic directive. The temporary partial results are stored in local private variables. The shared variable that contains the final results has been replicated to avoid using critical regions as they impose serialization in the access to these variables. At the beginning these private variables were stored in the stack, but sometimes the stack was overflowed due to the size of these variables and the final solution has been to use the heap, even if it is a less efficient solution.

FSL-C-CUDA: This version uses a completely different and add-hoc reimplementation of the "specialized" version of the FSL_delta method, using the CUDA-C language to exploit the huge amount of low level parallelism available in the current GPUs. The main difficulty of programming a GPU is to correctly map the inherent application parallelism onto the parallel hardware available in the GPU, to optimize the access of the computing threads to internal memory. The GPU board attached to the PCI-X bus acts as a coprocessor, so only the routines computationally more expensive are executed on the GPU as so called kernels. While the main part of the program and data remains in the host, the necessary input and output data for the kernels need to be transferred through the PCI-X bus. In our case, the specialized main time loop has been splitted into two parts, each one being implemented as a different kernel. The intermediate results of the first kernel are used in the second one. These results are not transferred to the host, but kept in the GPU memory for efficiency. The threads running the first loop do not need explicit synchronization, as there are no dependencies between channels. In the second loop, a reduction operation has been implemented

[6]http://www.openmp.org

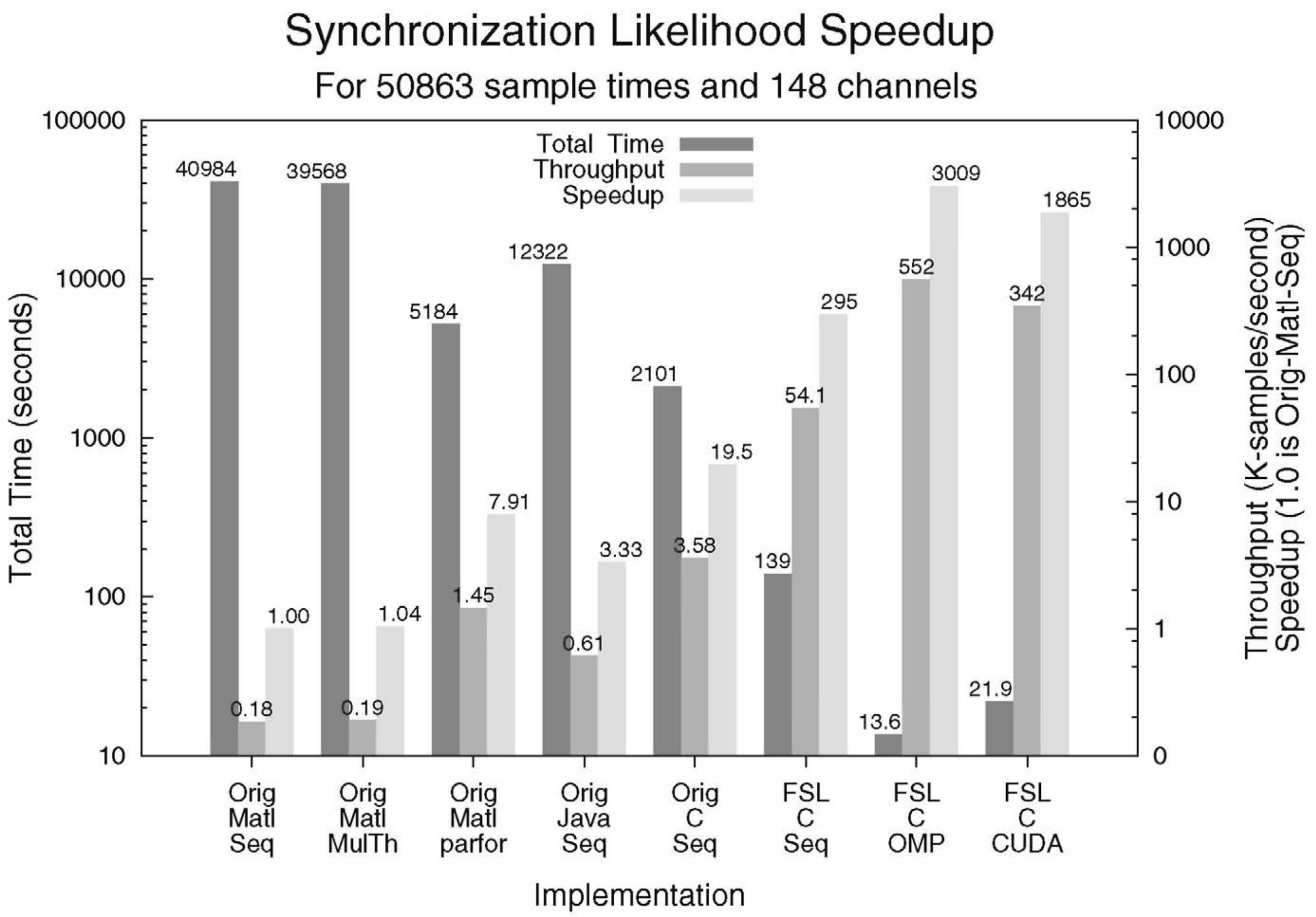

Fig. 2 Fixed dataset. Speed-up factor for the different implementations with respect to a non-optimized MATLAB implementation of the SL algorithm

using the shared memory (which is limited in size but very fast, even if it is explicitly controlled by the programmer) and the registers (to reuse the data previously read), accumulating the final partial results in a shared variable that is sent back to the host at the end. But the use of this reduction impose severe restrictions in parallalelization, as synchronization barriers must be used to ensure that results are correct. In the GPU each thread is in a block and multiple blocks are ordered in a grid. In this way the threads are organized in a logical hierarchy, which gets established when calling a kernel, when the dimensions of the grids and blocks are specified. In the first kernel, each block computes the embedding vectors for a different channel, and each thread of this block compute a different time-instant i. If there are more channels than blocks, each block computes several channels. In the second kernel, the recurrent states are computed. To compensate for the high latency of the global memory, Instruction Level Parallelism (ILP) is used ((Volkov 2010)). Both loops are unrolled to increment the available instructions that can be used to compensate for the high latencies. To improve the coalesced access to global memory the input data are transposed and aligned. To avoid using bifurcations, the beginning and end of the time series are not computed in the GPU, but elsewhere. As the GPU executes the same instruction for each warp, where a warp consists of a set of 32 threads, every bifurcation imposes the serial execution of each branch of the bifurcation.

Computer Hardware and SL Parameters

The SL parameters used in all the calculations are: w1 = 100, w2 = 410, L = 1, m = 10, p_{ref} = 0.049, speed = 1. These parameters determine the number of state vectors to be considered: $N = \lceil 2 \times (410 - 100) \times 0.049 \rceil = 31$, for an $Effective\ p_{ref} = 31/(2 \times (410 - 100)) = 0.05$.

In the case of the MATLAB calculations, the measures were taken without using the user interface (nodesktop and nosplash options) to avoid measuring the memory consumed by the interface. The MATLAB version used was R2012a 64-bit.

The calculations were carried out with double precision floating point (64 bits), on a dual processor Intel Xeon E5645 2.4 GHz with six cores each, totalling 12 cores with hyperthreading.[7]

Memory requirements (Fig. 3) were defined as the maximum resident set size, that is, the maximum amount of RAM memory used by the program during its execution.

For Fig. 6 the GPU used was a Nvidia GeForce GTX 580 with 512 cores, 1.5 GB of memory and a memory bandwidth of 192.4 GB/s.

The performance and memory requirements of the different algorithm implementations are reported in Figs. 2 and 3. A dataset with 148 channels and 50,863 samples per channel was used (sampling rate=254 Hz, duration=3 min 20 sec).

Results

Evaluation of Implementations

Figure 2 shows the processing time taken by the different SL implementations. A dataset consisting of 148 channels and 50,863 samples was used. Two MATLAB versions, the sequential (Orig-Matl-Seq) and the multithreading (Orig-Matl-MulTh) version, and the BrainWave version (Orig-Java-Seq) are quite slow, as they are able to process only 0.18, 0.19 and 0.61 Ksamples per second respectively. The translation of the original implementation to C (Orig-C-Seq) is much faster than these three versions, even if the same algorithm is used, processing 3.58 Ksamples per second. The main reason is that MATLAB is an interpreted language while C produces a compiled code. The explicitly parallel MATLAB version (Orig-Matl-parfor) has a speedup of nearly 8 with respect to the original MATLAB implementation, which we use as a reference. This is close to the nominal maximum of 8 cores imposed by the toolbox. Nevertheless it is still very slow, as it is able to process only 1.45 Ksamples per second. In contrasts, our sequential reimplementation of the Synchronization Likelihood (FSL-C-Seq) is able to process 54.1 Ksamples per second, being 295 times faster than the reference version. The GPU version (FSL-C-CUDA) implemented with CUDA on Nvidia graphic cards has a very high speedup factor of 1,865, compared to Orig-Matl-Seq and 6.32 compared to FSL-C-Seq, being able to process 342 Ksamples per second using an inexpensive graphics card. The version for shared memory computers (FSL-C-OMP) using OpenMP on a computer with 12 cores, presents an even higher speedup of 3,009 compared to Orig-Matl-Seq and 10.2 compared to the sequential FSL (FSL-C-Seq), reaching 552 Ksamples per second.

Figure 3 shows the memory requirements of the different implementations for the same dataset, with 148 channels and 50,863 samples. The explicitly parallel MATLAB version (Orig-Matl-parfor) requires significantly more memory, 58 % more, than the sequential MATLAB version. All the MATLAB implementations seem to use large amounts of memory in the form of auxiliary matrices of size proportional to the input data. This memory allocation is present for as long as the process is running. BrainWave (Orig-Java-Seq) is more memory efficient.

FSL-C-Seq is an autonomous program which highly optimizes the use of the memory. In fact, it has been designed to be able to process a data stream. It does not require loading all the data into memory, but just the necessary data to compute the distances in the surrounding double inclusion interval around each time point. All our versions (FSL-C-Seq, FSL-C-OMP and FSL-C-CUDA) require less than 1 % of the memory needed by the sequential MATLAB version, Orig-Matl-Seq, and are not limited by the dataset length as they only load into memory the data required to compute the distances needed for each time point. The FSL-C-OMP version only needs marginally more memory than the FSL-C-Seq version, as it uses private variables for the working threads. The CUDA version differs from the other ones in that it not only uses system memory but also video card memory. For this particular dataset 79MB of system memory plus 146MB of video card memory were used.

Scalability of Implementations

Next, we explore the scalability of the three novel implementations. First the scalability of the sequential version, FSL-C-Seq, is analyzed by gradually increasing the amount of data and/or the number of channels. Then, the scalability of the two parallel versions, using CUDA (FSL-C-CUDA) and OpenMP (FSL-C-OMP), is characterized using FSL-C-Seq as reference.

The previous dataset, with 148 channels and 50,863 samples per channel (sampling rate=254 Hz, duration=3 min 20 sec) is denoted with as an asterisk in Figs. 4, 5 and 6. Figures 4, 5 and 6 show the scalability of the sequential, OpenMP parallel and GPU parallel version respectively, with the amount of data increasing from 4K to 1024K and the number of channels ranging from 4 to 1024. To calculate the speedup factor, Fig. 4 uses as reference the smallest dataset with 4 channels and 4K data. The reference used in Figs. 5 and 6 is the sequential version with the same number of channels and samples.

[7]When hyperthreading technology is in use, each real core simulates two logical cores

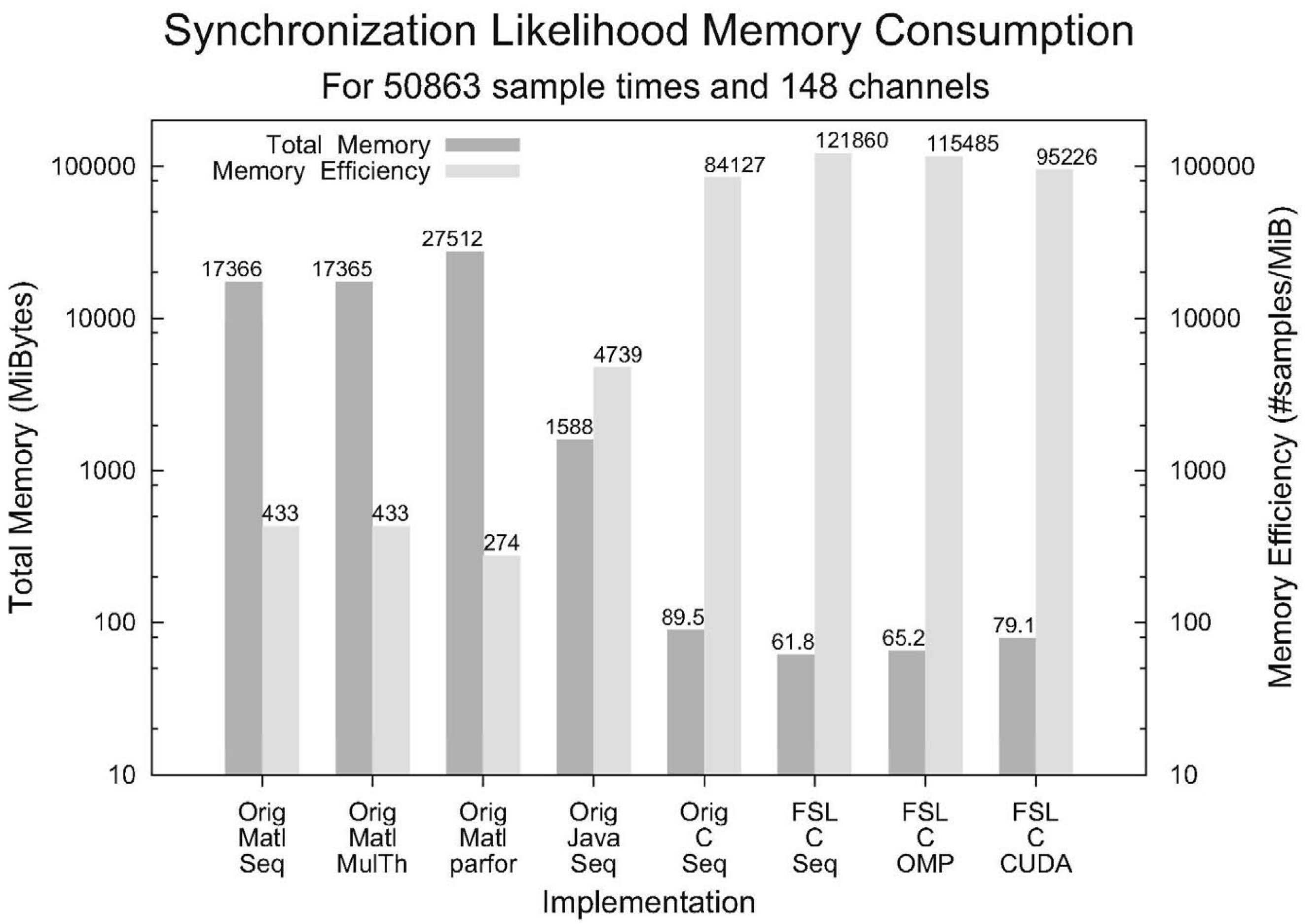

Fig. 3 Memory requirements for the different implementations for a given dataset

Figure 4 shows how the Sequential FSL version scales as a function of the number of samples per channel and number of channels. The speedup factor on the y-axis is defined as the ratio of number of processed samples per time unit for a given dataset to the same quantity for the reference dataset with 4 channels and 4K samples per channel. The fact that the lines are roughly constant indicates that sequential version scales linearly with the number of samples per channel. In contrast, the speedup factor decreases with increasing number of channels as the algorithm complexity in the second sub-loop (Fig. 1) is quadratic with respect to the number of channels.

The sequential MATLAB implementation, Orig-Matl-Seq, scale also linearly with respect to the number of samples per channel. The difference is that it scales linearly also with respect to number of channels. This analysis clarifies the behaviour of the OpenMP and CUDA parallel versions.

Figure 5 shows the scalabitly of the Open MP implementation. As seen in the figure, the amount of data has much less influence on the performance than the channel number, as could be expected after the results of the sequential scalability analysis. They are two different regimes. If the number of channels is small the maximum speedup is not achieved because there is not enough processing demand for all the cores. In this case, increasing the amount of data improves the speedup factor as it increases the amount of processing to be carried out by the underlying hardware. When the number of channels increases the speedup is limited by the number of cores.

Figure 6 shows the scalability of the CUDA parallel FSL version with on GPU GeForce GTX 580 with respect to the sequential version. The behaviour of this version is quite different from that the OpenMP version, as the amount of data greatly influences the performance. If there are few samples per channel the speedup as the amount of processing assigned to each GPU core is small. For larger number of samples per channel the speedup factor increases especially for an intermediate number of channels. For a large number of channels the speedup decreases as the data structures that the algorithm uses are rather large and no longer fit into the small data caches of the GPUs.

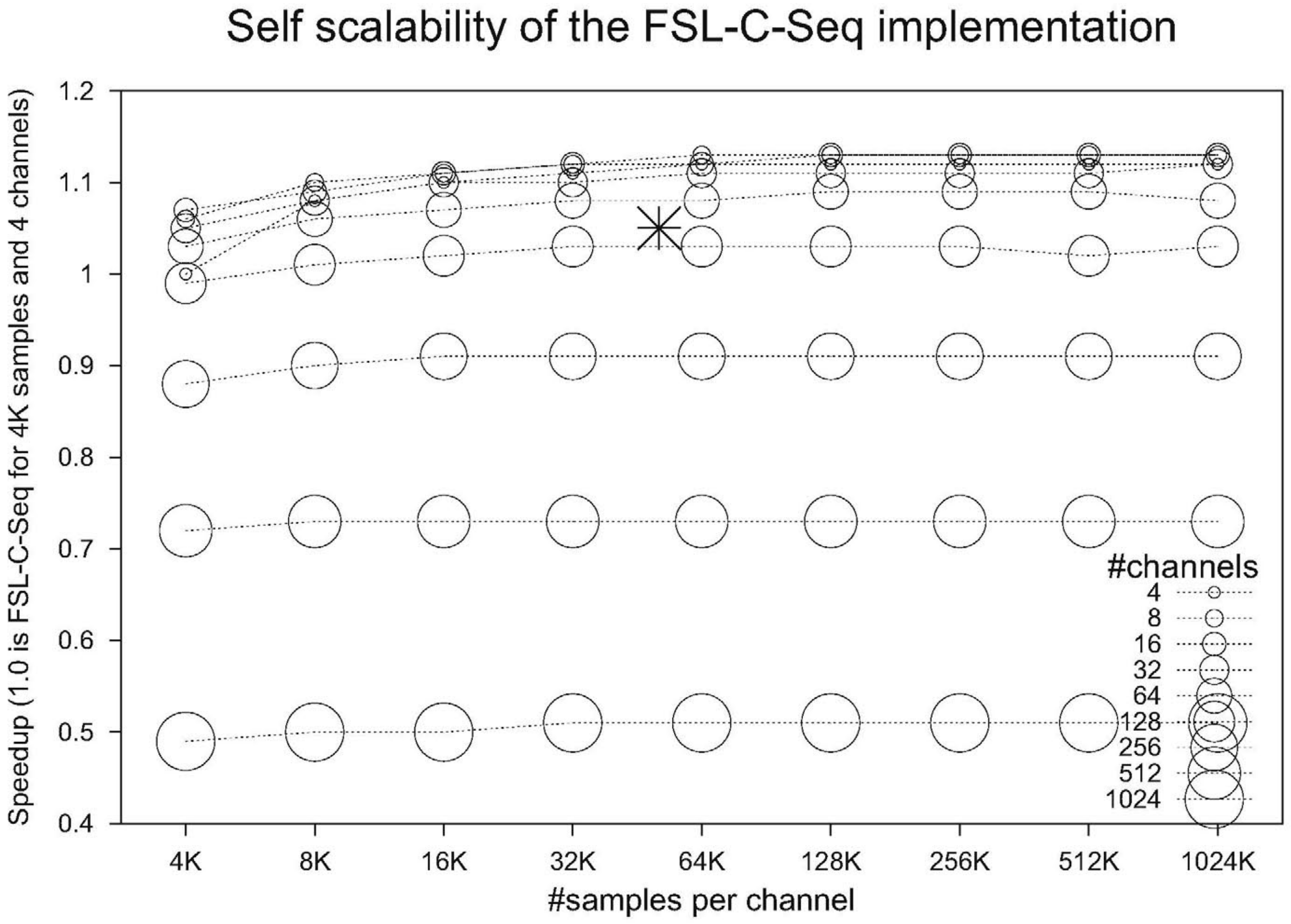

Fig. 4 Scalability of the Sequential FSL implementation for different dataset sizes

Computation of the Partial SL

The ability to obtain and store not only the global SL but also a partial SL (using the new parameter T) allows studying the dynamics of the system by observing how the SL indices evolves over time. Figure 7 shows, for the same real dataset of 148 channels and 50,863 samples used in Fig. 2 and 3, examples of partial SL matrix for the averaging periods 1, 16, 256 and 4096, and the global SL matrix. Each matrix pixel shows the averaged SL between each two channels over the period. We observe a consistent pattern that becomes fuzzier as the period (T) of the partial SL becomes bigger.

As our implementation also computes the Synchronization Entropy (Hs) (Stam et al. 2003), the Fig. 7 also shows on the background the average, standard deviation, maximum and minimum values of the Hs as a function of the averaging interval. As the averaging period becomes bigger, the averaged SL values are presumably more precise, but the higher frequency information is lost.

Validation of the Implementation

We tried to keep the Fast SL algorithm as compatible as possible to the original algorithm. One difficulty is that the original publication does not specify how calculations should be carried out at the beginning and end of the data series. So our FSL implementation is not only different from the original one due to the introduced simplifications, optimizations and the parallelization of the code, but also because the beginning and end of the data series may be treated differently.

This is the only reason why the FSL results are slightly different to those of the original implementation. In fact the FSL_delta "specialized" version (the one applied to the central segment of the data) produces exactly the same numerical results than the `sync_sa.m` MATLAB script used as representative of the original algorithm.

Some kind of validation is necessary to verify that the results of FSL remain equivalent to those of the original implementation. To our knowledge there is no reference

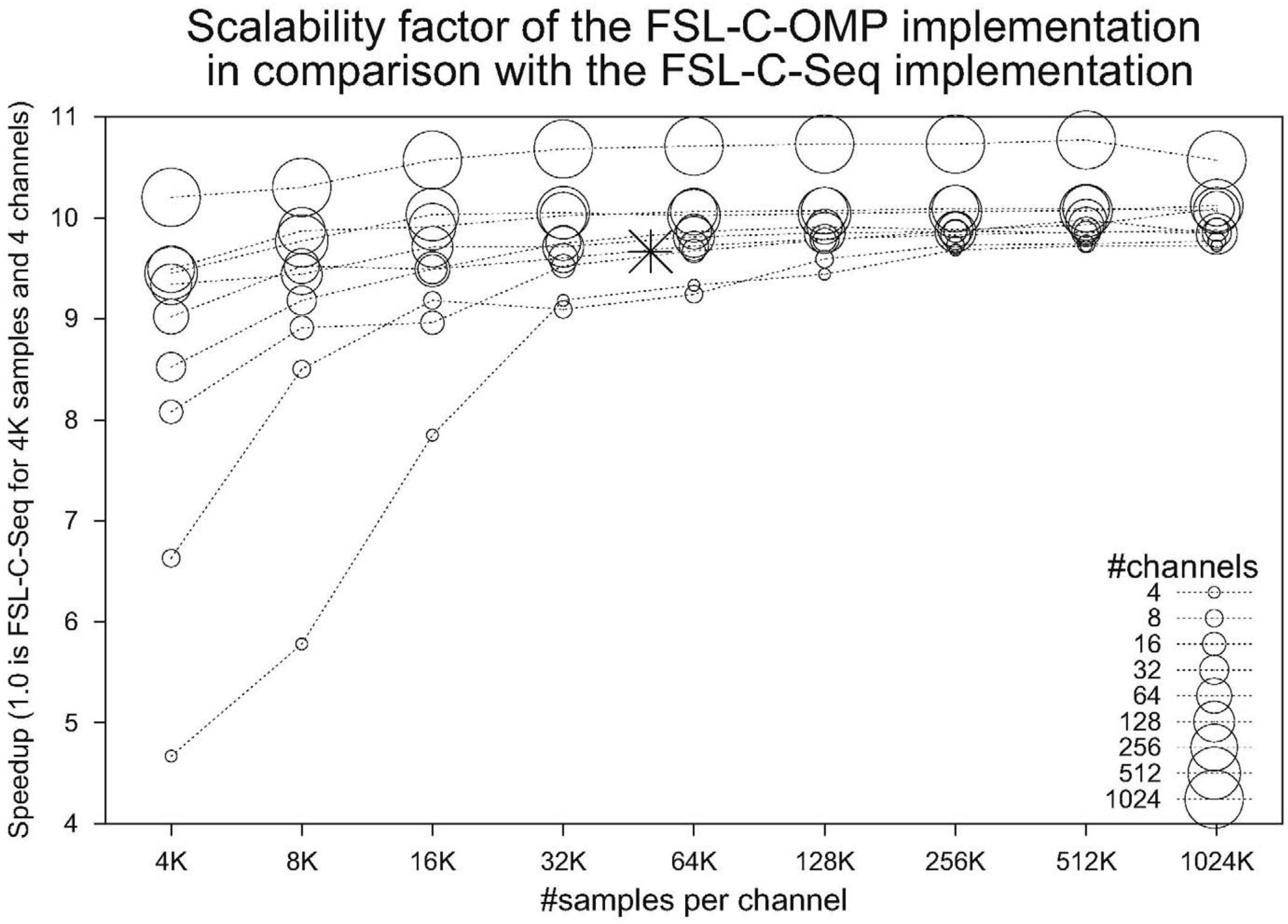

Fig. 5 Scalability of the OpenMP FSL implementation for different dataset sizes with respect to the sequential version

data set to validate the Synchronization Likelihood algorithm with. So we opted for reproducing analysis of a Henon system as in the Fig. 1 of Stam and Van Dijk (2002), as it is well described, easy to reproduce and summarizes a broad spectrum of cases since the degree of synchronization is changed smoothly.

Figure 8 shows the results obtained with FSL for this example. Each of the two curves consists of 101 points ranging from a Coupling Strength between 0 and 1. Represented is the average and standard deviation of the SL between 2 time series containing 4096 samples each. The calculation was repeated 10 times for different realizations generated from different seeds. A total of 8,273,920 samples were processed in 66 seconds.

This plot matches exactly the shape obtained in Stam and Van Dijk (2002). The number of points considered for the present figure is 10 times larger than for the original one. The difference between our results and those given by the original implementation can be measured as an accumulated relative difference[8] of 0.391 % for the "Identical, $B = 0.3$" curve and 0.451 % for the "Non-Identical, $B = 0.1$" curve.

In addition, in order to also validate our algorithm with empirical data, we applied the SL to a freely available dataset of Magnetoencephalography (MEG) recordings available from the Human Connectome Project. This dataset includes high quality MEG scans from 14 healthy adults (all members of monozygotic twin pairs) collected at rest. The sampling rate was 2034.51 Hz. The scanner had 248 magnetometer channels and 23 reference channels (for more details about the dataset, see "HCP MEG Scan Protocol Details" in the Human Connectome Project

[8] Accumulated Relative Difference of two curves (a and b) for values V in P points:
$ARD = \sum^P |V_a - V_b| / \sum^P (V_a + V_b)/2$

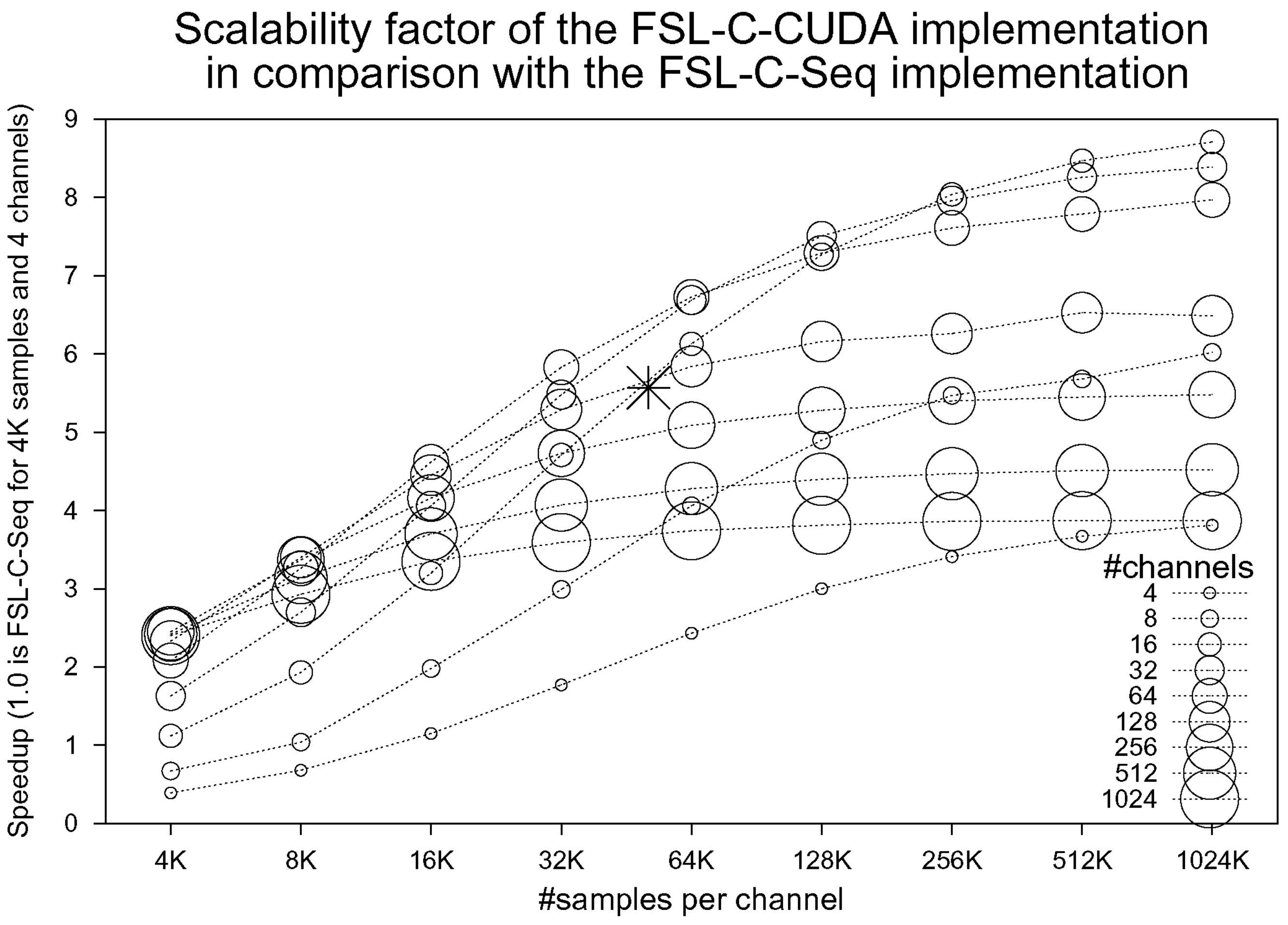

Fig. 6 Scalability of the CUDA FSL implementation for different datasets

webpage). Three of these MEG recording were employed to validate our algorithm. We download the raw, unprocessed MEG data in 4D Neuroimaging format corresponding to three subjects. Subsequently, and prior to functional connectivity analysis (SL analysis), the three records were visually inspected by an experienced investigator, excluding visible blinks, eye movements or muscular artifacts from the data. For each of the three subject, 5 epochs free of artifacts of 4096 points (of around one second in lenght) and 248 channels, of resting state activity were selected. Subsequently, the SL algorithm was applied to the 5 extracted artifact-free epochs for each of the three subject. The SL was calculated for each of the 5 one-second epochs with 248*247/2 channel pairs for each subject. The SL algorithm was calculated in two ways: a) using the FSL implementation and b) using the SL implementation available in the BrainWave software package. For all channel pairs and epochs, the relative differences in SL values between both methods (a) and b)) never exceeded 1 %.

Discussion

The novel SL algorithm implementation described in this paper has the following features. 1) Boundary segments are processed adaptively so that SL is computable from short time series. 2) Computations can be done with partial data segments only, without the need to load the whole dataset into memory, which allows for data streaming and processing of datasets of unlimited size. 3) The Synchronization Entropy (Hs) and the partial SL can be computed efficiently.

In the present work we have shown that optimizing the implementation of the algorithm used to estimate a connectivity measure, the SL, can increase the processing speed and decrease the amount of required memory by orders of magnitude. The combined effect of redesigning the algorithm and porting it from MATLAB to C increases speed by a factor of approximately 300 with respect to the originally published implementation. If one adds

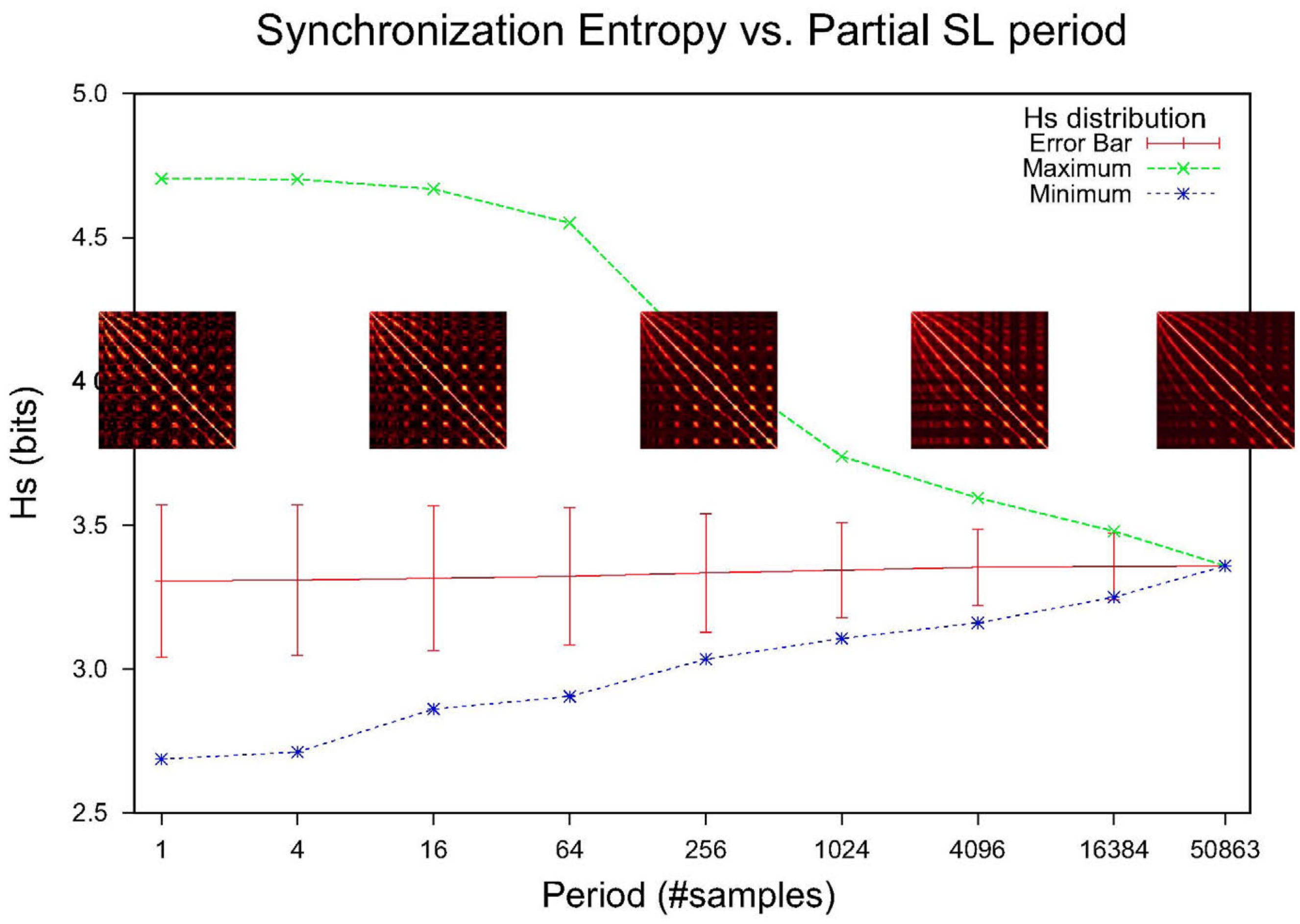

Fig. 7 Partial SL as a function of the averaging period

parallelization, the combined speedup factor is around 3,000 with 12 cores, and around 2,000 using an inexpensive GPU, a GeForce GTX 580 with 512 cores (Fig. 2). Likewise the optimization yields a decrease of a factor of approximately 200 in the amount of memory needed to perform the computations (Fig. 3).

The computational time for the novel sequential implementation increases linearly with the number of samples (Fig. 4). The novel parallel implementation, with an OpenMP architecture, speeds-up processing by a factor approximately equal to the number of processors (Fig. 5). On the other hand, if the more affordable CUDA architecture is employed, the speed-up processing is also of an order of magnitude for the considered representative case, but the scaling with the number of sensor pairs is less efficient than with OpenMP (Fig. 6). Additionally, CUDA programming and maintenance is complex.

To summarize, as shown in Figs. 2 and 3, the new FSL implementation is much faster and memory efficient than the original version and it is therefore our recommended option. Which type of parallelization is advantageous to use depends on the hardware available.

Therefore, both the new algorithm and the parallelization of the implementation, provide a huge increase in the efficiency of the computation of the SL. This should allow the undertaking of more ambitious connectivity analyses than the ones afforded by the current implementations. At present, the connectivity analysis is one of the most computationally costly steps of electromagnetic signal analysis. The current work presents a new optimized implementation which should widen the applicability of this type of analysis. The amount of time needed to compute the SL for a typical single participant recording with the original implementation (Fig. 2) is approximately 10 hours. If we take into account that we may need to do this analysis for 10-50 participants times 3-4 experimental conditions, and this may have to be repeated several times as we refine our analysis, we start to appreciate the computational burden imposed by the speed of the original implementation on the analysis. An increase in speed of 2 or 3 orders of magnitude is therefore very welcome.

With the present developments, a connectivity measure among more channel pairs can be calculated without the analysis becoming computationally prohibitive. Also a

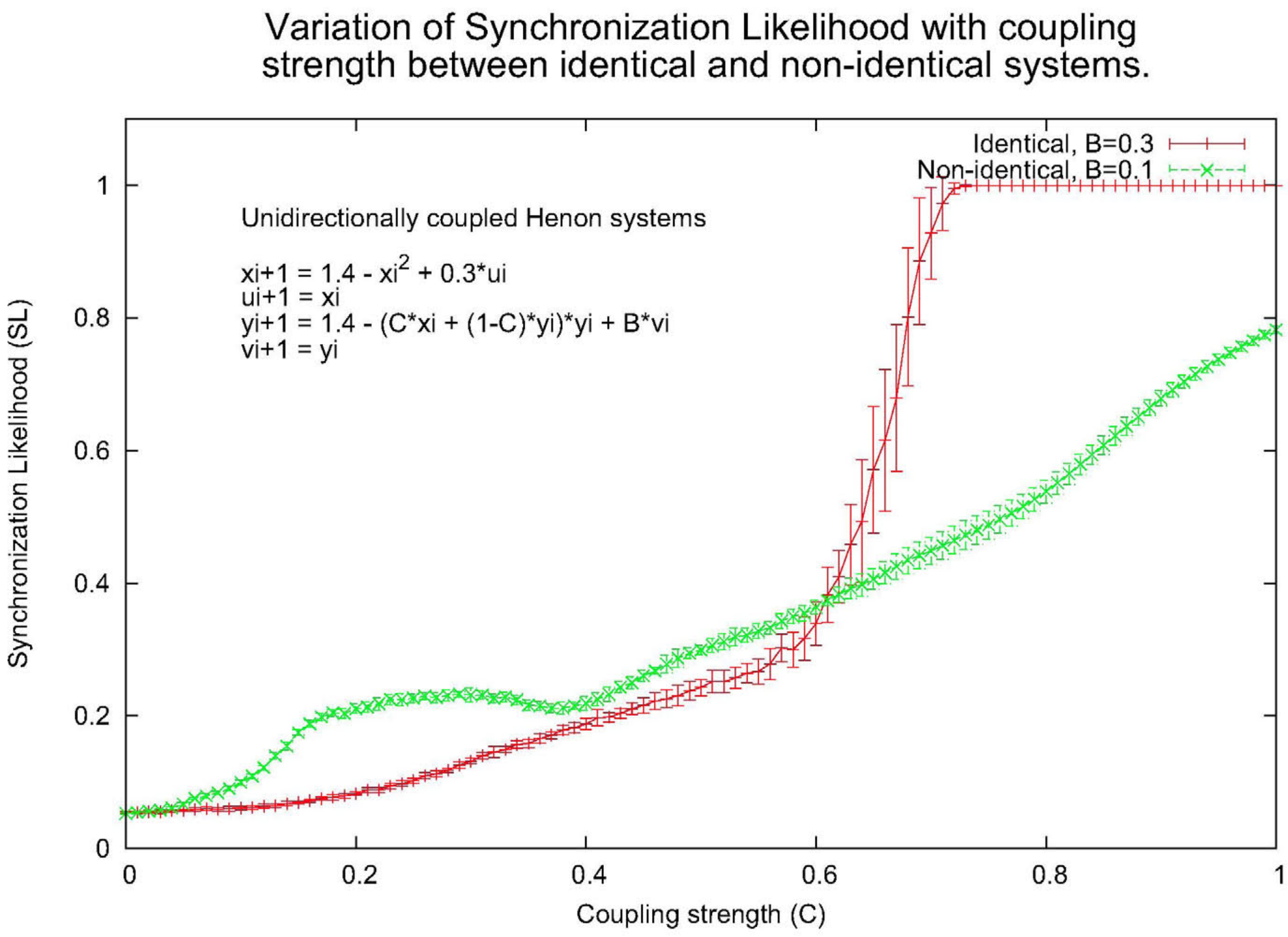

Fig. 8 Reproduction of Henon figure from Stam and Van Dijk (2002)

higher temporal and spatial resolution can be afforded. Furthermore, it should be feasible to use permutation testing in a wider set of circumstances than previously. Also the fact that speed has been greatly improved makes real-time analysis of the signals, with its potential for, for example, brain computer interfacing, within reach. An added advantage of the implementation here described is that they are much less memory demanding. This makes previously unfeasible analyses of long duration datasets, like those obtained from some epilepsy studies, treatable. In addition, it should now possible to calculate the SL between a large number of time-series, such as those corresponding to source-reconstructed time-courses of EEG/MEG data.

Future work could include the application of these optimization strategies to other connectivity measures where applicable, and the testing and adaptation of these new algorithms for real-time applications such as brain computer interfacing.

In conclusion, connectivity analyses represent a key ingredient in current neuroimaging signal processing. One of the most widely used indices is generalized synchronization, which is one of the types of analysis which is more computationally costly. The current work presents a new optimized implementation which should widen the applicability of this approach.

Information Sharing Statement

The source code for FSL (RRID:nif-0000-00305) is available under the Lesser GNU Public License (LGPL). The tarball is hosted by the Center for Biomedical Technology (CTB) under the HERMES project (RRID:nlx_155770, http://hermes.ctb.upm.es/resources/FSL/). Anonymous read access to the source code is enabled.

Acknowledgments This research was supported in part by the Spanish Ministry of Economy and Competitiveness (National project TEC2012-38453-C04 and Grant PSI2010-22118).

References

Acharya, A., Kar, S., Routray, A. (2010). Phase synchronization based weighted networks for classifying levels of fatigue and sleepiness. In *2010 international conference on systems in medicine and biology. IEEE* (pp. 265–268).

Ahmadlou, M., Adeli, H., Adeli, A. (2012). Fuzzy Synchronization Likelihood-wavelet methodology for diagnosis of autism spectrum disorder. *Journal of Neuroscience Methods*, 1–7.

Bajo, R., Maestú, F., Nevado, A., Sancho, M., Gutiérrez, R., Campo, P., Castellanos, N.P., Gil, P., Moratti, S., Pereda, E., Del-Pozo, F. (2010). Functional connectivity in mild cognitive impairment during a memory task: implications for the disconnection hypothesis. *Journal of Alzheimer's Disease : JAD*, *22*(1), 183–193.

Betzel, R.F., Erickson, M.A., Abell, M., O'Donnell, B.F., Hetrick, W.P., Sporns, O. (2012). Synchronization dynamics and evidence for a repertoire of network states in resting EEG. *Frontiers in Computational Neuroscience*, *6*, 74.

Buldú, J.M., Bajo, R., Maestú, F., Castellanos, N., Leyva, I., Gil, P., Sendiña Nadal, I., Almendral, J.a., Nevado, A., Del-Pozo, F., Boccaletti, S. (2011). Reorganization of functional networks in mild cognitive impairment. *PloS one*, *6*(5), e19584.

Buzsáki, G., & Draguhn, A. (2004). Neuronal oscillations in cortical networks. *Science (New York, N.Y.)*, *304*(5679), 1926–9.

Buzug, T., Pawelzik, K., von Stamm, J., Pfister, G. (1994). Mutual information and global strange attractors in Taylor-Couette flow. *Physica D: Nonlinear Phenomena*, *72*(4), 343–350.

Calmels, C., Hars, M., Holmes, P., Jarry, G., Stam, C.J. (2008). Nonlinear EEG synchronization during observation and execution of simple and complex sequential finger movements. *Experimental Brain Research*, *190*(4), 389–400.

Castellanos, N.P., Paúl, N., Ordóñez, V.E., Demuynck, O. Bajo, Campo, P., Bilbao, A., Ortiz, T., Del-Pozo, F., Maestú, F. (2010). Reorganization of functional connectivity as a correlate of cognitive recovery in acquired brain injury. *Brain: A Journal of Neurology*, *133*(Pt 8), 2365–2381.

Dagum, L., & Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, *5*(1), 46–55.

David, O., Cosmelli, D., Friston, K.J. (2004). Evaluation of different measures of functional connectivity usinga neural mass model. *NeuroImage*, *21*(2), 659–73.

Fries, P. (2005). A mechanism for cognitive dynamics: neuronal communication through neuronal coherence. *Trends in Cognitive Sciences*, *9*, 474–480.

Friston, K.J. (1994). Functional and effective connectivity in neuroimaging: A synthesis. *Human Brain Mapping*, *2*, 56–78.

Guggisberg, A.G., Honma, S.M., Findlay, A.M., Dalal, S.S., Kirsch, H.E., Berger, M.S., Nagarajan, S.S. (2008). Mapping functional connectivity in patients with brain lesions. *Annals of Neurology*, *63*, 193–203.

Montez, T., Linkenkaer-Hansen, K., Van Dijk, B.W., Stam, C.J. (2006). Synchronization likelihood with explicit time-frequency priors. *NeuroImage*, *33*, 1117–1125.

Niso, G., Bruña, R., Pereda, E., Gutiérrez, R., Bajo, R., Maestú, F., Del-Pozo, F. (2013). HERMES:towards an integrated toolbox to characterize functional and effective brain connectivity_ ISACM. In *International Society for the Advancement of Clinical Magnetoencephalography*, (p. 38453).

Pereda, E., Quiroga, R.Q., Bhattacharya, J. (2005). Nonlinear multivariate analysis of neurophysiological signals. *Progress in Neurobiology*, *77*(1-2), 1–37.

Pijnenburg, Y.A.L., V D Made, Y., Van Cappellen Van Walsum, A.M., Knol, D.L., Scheltens, P., Stam, C.J. (2004). EEG synchronization likelihood in mild cognitive impairment and Alzheimer's disease during a working memory task. *Clinical Neurophysiology*, *115*, 1332–1339.

Posthuma, D., de Geus, E.J.C., Mulder, E.J.C.M., Smit, D.J.A., Boomsma, D.I., Stam, C.J. (2005). Genetic components of functional connectivity in the brain: the heritability of synchronization likelihood. *Human Brain Mapping*, *26*(3), 191–8.

Singer, W. (1999). Neuronal synchrony: a versatile code for the definition of relations?. *Neuron*, *24*(1), 49–65,111–25.

Singer, W. (2013). *Cortical Dynamics Revisited. Trends in Cognitive Sciences*, (pp. 1–11).

Stam, C.J., Breakspear, M., van Walsum, A.-M.v.C., van Dijk, B.W. (2003). Nonlinear synchronization in EEG and whole-head MEG recordings of healthy subjects. *Human Brain Mapping*, *19*(2), 63–78.

Stam, C.J., De Haan, W., Daffertshofer, A., Jones, B.F., Manshanden, I., Van Cappellen Van Walsum, A.M., Montez, T., Verbunt, J.P.A., De Munck, J.C., Van Dijk, B.W., Berendse, H.W., Scheltens, P. (2009). Graph theoretical analysis of magnetoencephalographic functional connectivity in Alzheimer's disease. *Brain: A Journal of Neurology*, *132*, 213–224.

Stam, C.J., & Van Dijk, B.W. (2002). Synchronization likelihood: an unbiased measure of generalized synchronization in multivariate data sets. *Physica D: Nonlinear Phenomena*, *163*, 236–251.

Takens, F. (1981). Detecting strange attractors in turbulence. *Dynamical systems and turbulence Warwick 1980*, *898*, 366–381.

Theiler, J. (1986). Spurious dimension from correlation algorithms applied to limited time-series data.

Tononi, G., Sporns, O., Edelman, G.M. (1994). A measure for brain complexity: relating functional segregationand integration in the nervous system. *Proceedings of the National Academy of Sciences of the United States of America*, *91*(11), 5033–7.

Varela, F., Lachaux, J.-p., Rodriguez, E., Martinerie, J. (2001). The brainwave_ Phase synchronization and large-scale integration. *Nature reviews. Neuroscience*, *2*.

Volkov, V. (2010). Better performance at lower occupancy. *Proceedings of the GPU technology conference, GTC*, *10*.