Contents lists available at ScienceDirect

Journal of Neuroscience Methods

journal homepage: www.elsevier.com/locate/jneumeth

Parallel calculation of multi-electrode array correlation networks

Pedro Ribeiro[a,*], Jennifer Simonotto[b,c], Marcus Kaiser[b,c,d], Fernando Silva[a]

[a] Universidade do Porto, Faculdade de Ciências, CRACS Research Center, Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
[b] School of Computing Science, Newcastle University, Claremont Tower, Newcastle-upon-Tyne NE1 7RU, UK
[c] Institute of Neuroscience, Newcastle University, Newcastle-upon-Tyne NE2 4HH, UK
[d] Department of Brain and Cognitive Sciences, Seoul National University, Shilim, Gwanak, Seoul 151-747, Republic of Korea

ARTICLE INFO

Article history:
Received 27 January 2009
Received in revised form 3 August 2009
Accepted 4 August 2009

Keywords:
Correlation networks
Parallel computing
Condor
Neuroimaging
Optical imaging
Time series analysis

ABSTRACT

When calculating correlation networks from multi-electrode array (MEA) data, one works with extensive computations. Unfortunately, as the MEAs grow bigger, the time needed for the computation grows even more: calculating pair-wise correlations for current 60 channel systems can take hours on normal commodity computers whereas for future 1000 channel systems it would take almost 280 times as long, given that the number of pairs increases with the square of the number of channels. Even taking into account the increase of speed in processors, soon it can be unfeasible to compute correlations in a single computer. Parallel computing is a way to sustain reasonable calculation times in the future. We provide a general tool for rapid computation of correlation networks which was tested for: (a) a single computer cluster with 16 cores, (b) the Newcastle Condor System utilizing idle processors of university computers and (c) the inter-cluster, with 192 cores. Our reusable tool provides a simple interface for neuroscientists, automating data partition and job submission, and also allowing coding in any programming language. It is also sufficiently flexible to be used in other high-performance computing environments.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Due to new experimental techniques, the amount of neural activity data from a single experiment has increased in recent years. Whereas early recordings involved single electrodes or tetrodes (a set of four electrodes), novel setups use grids of electrodes in intracranial recordings (ECoG) on the global scale or multi-electrode arrays (MEAs) on the local scale. Nowadays, MEAs record around 60 channels (or more) of 16-bit data sampled at typical rates of 25 kHz, more than ample for both local field potentials and spikes from local neurons to be recorded. However, setups with 1000 electrodes are already implemented and will become more common in the next years. In addition, techniques of optical imaging or MRI neuroimaging provide millions of channels corresponding to image pixels or voxels, respectively. The challenges of data management and analysis have led to several neuroinformatics (Eckersley et al., 2003) projects including the International Neuroinformatics Coordinating Facility (INCF), the FIND Matlab toolbox for multiple-neuron analysis (Meier et al., 2008), and the CARMEN e-science project (Watson et al., 2007; Smith et al., 2007; Eglen et al., 2007) to which this tool is related.

Correlation of activity patterns between all pairs of electrodes is a useful measure to analyze neural activity data. Such comparisons can yield a correlation network where an edge between nodes (here: electrodes) is either weighted using the mean correlation value (as in Fig. 3) or binary (being 1 if the correlation is above a given threshold or 0 otherwise, as in Fig. 4). Since MEAs can record for long periods of time, there may be non-stationarity in the time series. To cope with this, we use sliding windows: we evaluate the correlation on an interval of time t, then slide that interval by k positions and evaluate again, producing a series of correlation matrices for each interval that correspond to different instantaneous networks. Such correlation networks, also called functional networks (Sporns et al., 2004), have shown specific deviations in the global network organization in schizophrenia, Alzheimer's, and epilepsy patients (Micheloyannis et al., 2006; Stam et al., 2007; Ponten et al., 2007) as well as for aging in human subjects (Salvador et al., 2005; Achard and Bullmore, 2007). It can be expected that functional connectivity also relates to classes of networks at the local scale of MEA recordings. Given functional connectivity, network analysis (Albert and Barabási, 2002; Costa et al., 2007) can be used to assess topological changes over time.

Correlation can be coded in a variety of programming languages, but within the neuroscience community Matlab is an often used tool, as it provides a wide array of ready-coded algorithms and also allows the user to customize code in a relatively easy way. The algorithm we devised to calculate the correlations requires a computing time that is proportional to the square of the number

* Corresponding author. Tel.: +351 220402925; fax: +351 220402950.
E-mail addresses: pribeiro@dcc.fc.up.pt (P. Ribeiro), jennifer.simonotto@ncl.ac.uk (J. Simonotto), m.kaiser@ncl.ac.uk (M. Kaiser), fds@dcc.fc.up.pt (F. Silva).

0165-0270/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.jneumeth.2009.08.001

of channels C (all-to-all) and also proportional to the number of measured values V (that is, the order of complexity is $O(C^2V)$). As we increase the time interval t, or the size of slide k, the time needed to calculate increases significantly. Current MEAs and typical choices of the parameters can lead to spending many hours, even days, to calculate correlation networks on normal serial commodity computers. We can expect that larger MEAs will appear, with the number of channels in the order of a 1000 and higher temporal resolution, thus leading to an increased number of values in the time series. This can make it unfeasible to calculate features like the correlations networks in real time.

One way we can improve the time needed to complete the calculations is by using the power of parallel computing, using multiple processors to jointly perform all the computation needed. In our particular situation we had two computational resources available: a single cluster with 16 processors, and a big pool of more than two hundred university computers. Both shared the same job control system: Condor (Thain et al., 2005), but others could have also been used. As we will see in Section 3, the correlation problem we were dealing with has inherent data parallelism, and therefore is a good candidate to parallel calculation.

Python has been proposed as a means to speed up calculation time (Spacek et al., 2008), including parallel computing via Open MPI (Ince et al., 2009). However, existing algorithms need to be ported to this new language, which may not be feasible for large amounts of legacy code. A first step towards automatic code transfer are recently developed compilers from Matlab, the currently most widely used environment for data analysis, to Python (Jurica and van Leeuwen, 2009).

A problem with parallelization is that the typical scientist not coming from a computer science background is used to normal serial tools and the amount of work involved getting up to speed for running even the simplest application in parallel can deter one from using it (Pancake, 2003). Factors prohibiting the use of parallel computing range from the need of recoding programs using the message passing interface (MPI) to availability of computing clusters and getting permission to use such systems. There are many possible applications for data parallelism in the realm of neural science (Oppenheim et al., 1992; Martino et al., 1994) (and also outside of it). The scientists may want to run code in a scalable way in order to test some new idea. There is a need, then, for more seamless ways of providing parallel computing power to the neuroscience community.

We could have used an existing parallel software framework, such as MPI, but that would require the need for the scientist himself to understand the mechanisms and syntax of MPI, would require the usage of a language with an MPI interface, and it would not be suitable to use with Condor (MPI needs dedicated clusters, since all the processes are running at the same time and communicating between themselves). We could have used a specialized platform for data intensive tasks like Bialecki et al. (2009), but that would require that our existent code would have to be changed and the the cluster would have to support this specific file system.

Although we had available a large Condor platform, we decided to construct a more general and reusable tool to run parallel applications on multiple computing platforms. Our target applications are very specific: they invoke tasks that are inherently parallel, in the sense that parallelism arises from the split of data among the compute nodes. Each node then executes the same task on its specific given data and the global result can be obtained by aggregating the results of all local computations. Splitting of data can be fully automated or guided by parameters provided by the user. Data parallelism has the potential to induce an almost linear speedup of the computation and makes it possible to deal with data intensive applications in an efficient and scalable way. In addition, we wanted the tool to be easily ported to similar environments that do not use Condor, but use other batch submissions systems, like Torque/PBS (Torque, 2009; PBS, 2009) or sun grid engine (SGE) (Gentzsch, 2001), which are widely used in clusters all around the world. Finally, we wanted the possibility for the user of writing the specific calculation code (in our case the correlation) in any given language, as long was it would be executable on the targeted computers. This way, the scientist can still use his familiar programming language, instead of having to learn a new language to be able to use an API for a specific parallel framework.

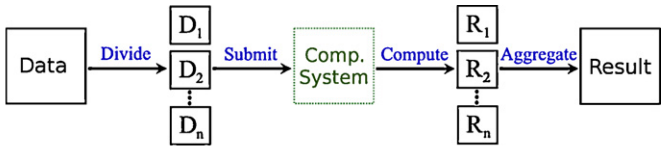

Fig. 1. Architecture workflow of Adapa. The data is divided into smaller partitions that are then submitted to the computation environment. A set of results is then obtained and aggregated to form the end global result.

The remainder of the paper is organized as follows. Section 2 briefly describes the proposed tool, by showing all its components and control flow. Section 3 describes how we used the tool in our specific application of calculating correlation networks. We use real and random data and show execution times in Condor and other environments. Section 4 concludes the paper, discussing the results and including some possible future work.

2. Materials and methods

In this section we describe the main design ideas of our framework, which we call Adapa (Automatic DAta PArallelism).[1] It aims to simplify the use of a cluster/grid environment by non-specialists and allows users to prepare, submit, compute and gather results. Adapa was developed in C++ in a modular way allowing for future reuse and extendability to new functions or other systems.

The basic workflow of Adapa is divided in four major phases, as illustrated in Fig. 1. A first preparation phase involves dividing data according to the user specifications into separate data files (called partitions) and creating job submission files to run on the system. The second phase is for job submission to the execution nodes. The third phase is the actual execution of the jobs, and the final step is to collect and aggregate the results.

Each application we want to run is called an experiment in the context of Adapa, and the file system is used to maintain a different directory for each one of them. So, one could have several experiments being computed by our tool and easily navigate between them by simply changing the directory.

We now explain the four phases of the workflow of Adapa in more detail.

2.1. Dividing input data

To be able to exploit the potential data parallelism of a specific experiment, it is vital to know how to divide the data into different partitions. The main idea is that different processors can compute different partitions at the same time, thus reducing the total time needed for the whole computation.

Knowing how to divide for the general case of any algorithm in an optimized way is very difficult, and perhaps even impossible(Mitra et al., 2000). However, Adapa tries to foresee the most usual division patterns arising in scientific applications and automates the division step, giving three basic division profiles:

[1] Adapa is also the name of a Babylonian mythical figure.

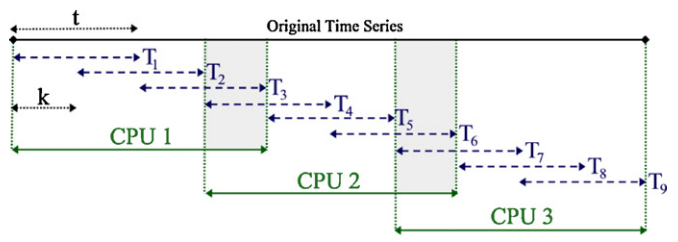

Fig. 2. Example of data division of nine intervals for three processors. Each interval has size t and differs from the last one by a slide of k positions. Each processor gets the same amount of data (three intervals each) and two "consecutive" processors share only a small amount of equal data (with exactly size $t - k$, depicted in the figure as shaded grey).

- **Equally sized independent partitions**: The data is divided into a series of equally sized, independent and consecutive partitions. So, if we have n values and we want p partitions, each one will roughly have n/p values. Adapa allows the user to do this kind of division either by specifying the desired number of partitions (where it automatically determines their size) of by specifying each partition size (where it automatically determines the total number of partitions);
- **Sliding window partitions**: The data is divided into a series of equally sized partitions of size t, each one obtained by sliding a window of size t by a determined number of values k (the user specifies t and k). Fig. 2 exemplifies this division scheme with nine partitions for three processors;
- **Customized partitions**: The user specifies the start and end points of each partition (overlapping is allowed). This adds extra flexibility in the case one wants a different division pattern and virtually allows any possible scheme. The user has to give the list of correspondent integer pairs and can potentially even use other programs to generate that list.

It should be noted that new automated division schemes can easily be added in the future, given the modular nature of our tool.

Besides indicating how to divide the data, the user can also specify the type of data used. Currently Adapa supports fixed-size (the user specifies the number of bytes per value) or one value per line (typical in ASCII files) data streams.

The end result of this step is a series of files with the data already divided.

2.2. Submitting data

To submit the data to a node for execution, we need to talk to a system that manages the computation work. In our case, we chose Condor as the initial specific system to interact with (presently we have already extended it to other batch systems, such as SGE or PBS/torque). In order to submit a job to Condor, one needs to prepare a submission file, indicating details such as the input, the output and what files we need to transfer to the execution node. Adapa generates this file automatically and then calls the Condor mechanism to put all partition jobs as a single independent job on the job queue, waiting to be executed.

There are several interfaces to Condor. We opted to use the simplest one, a command line interface, since it was the option that promised to have the better performance. The Grid ASCII Helper Protocol (GAHP, 2009) is a more general way of interacting with grid systems, available in Condor, but it poses several problems with scalability. One could also use BirdBath (Chapman et al., 2005) as a way to call Condor functions using Web Services and SOAP, but in our case it would mean that we would need to augment the functionality of the existing Condor daemons by installing the necessary software, which was not desirable as a readily running solution that required a minimum environment "tweaking" was sought after.

Adapa is prepared for two different scenarios: shared disk space among processors and no sharing at all. In the first case, no data in itself needs to be sent, since the processors can access the divided files directly (and the responsibility of managing this access lies outside our tool, that is on the chosen shared file-system).

In the second case, given that the data to be sent to each node can be potentially very large, Adapa provides functionality to automatically compress and send the data as well as to decompress at the node before calling the executable. One can use any compressor, as long as the commands for compressing and uncompressing are available by command line at each node.

Adapa also supports heterogeneous environments in the sense that it is possible to specify different executables to be run on different end systems. For example, one could provide an executable to be run for Windows 32-bit architecture and another for Linux 64-bit architecture.

2.3. Executing the task

The key component of the computation is the actual program that is going to process the data itself. In Adapa, the executable to be called is completely customizable in the sense that we just provide the executables we want to be run, written in any language we want. Condor provides several universes (e.g. `Vanilla`, suitable for any executable or `Java`, preferred for running Java applications), and Adapa gives the user the option to choose the universe in which the user wants to run the executable.

In order to communicate with the executable there are two main options: one could use the standard input to the program or use the command line arguments in which we call the executable. Adapa allows both. Each executable receives information on which files it has available (like size and name) by being fed an input file which was created during the division phase. Condor provides the mechanism to transfer all relevant files to the node where the computation is to be run and also takes care of giving our input to the program itself. With this, one only needs to specify to the executable the format in which this input will be given. The tool and the submission system take care of all the rest, guaranteeing that all jobs are correctly started on the correspondent processors.

2.4. Collecting results

The execution of jobs produces in the end a series of result files. The user can use Adapa to automatically concatenate those to a single file (the inverse of partitioning the data) or to specify a custom command that can apply more complex functions to the output (e.g. additional analysis). Another option is to simply do nothing to the result files, since they may already be in a format that we can use for further analysis and computations.

2.5. Usage, quality of service and fault tolerance

Adapa is currently used via the command line (a graphical user interface is planned). In order to define all the parameters for which we want our application to be run, we have to edit a series of documented text files. We can have several experiments in different stages of the execution, each one of them in an independent directory with its own configuration files specifying things like how to divide the data and what executable we have to run.

Adapa can be directly called for each specific phase. One can simply ask Adapa to: (a) divide the data, without actually submitting it (to preview the resulting division); (b) tell the tool to submit an already partitioned data; (c) collect the data; or (d) run all above steps.

At any time, Adapa provides ways to see what jobs are running (including running time), what jobs are still in the queue, to cancel the experiment or to specifically ask for a particular partition to be run again. In order to do that, the batch submission system interface is used.

Our tool also relies on the batch system to provide some quality of service. Instead of trying to do everything ourselves, we let it do what it is suited for, like deciding the best available resources, or automatically restarting a job which was compromised. Adapa only divides and creates the jobs; it is the high performance computing system that determines the best way to run them.

2.6. Availability and potential uses

Adapa is now on a functional prototype phase, where it can already be used by real scientists. After doing some more tests, creating a graphical user interface and the necessary documentation, we plan to make the tool available online, under a GPL license, to be available to anyone who wishes to use it[2] (Ribeiro et al., 2009).

The next section describes how we used Adapa to calculate correlations but as we saw in the previous sections the tool can really be used in other situations that exhibit data parallelism. All the user needs is: (1) a high performance resource (like a Condor Pool) to which Adapa can communicate with; (2) the data files; (3) an executable program in the desired platform able to calculate the result when fed with a chunk of data (a partition). Note that the executable can be written in any language, therefore allowing the user to resort to a familiar environment. Without significant code-level tinkering (besides being able to read from Adapa the information regarding a single partition), one can then easily parallelize the application. Our tool then takes care of the real more tedious and long task of physically splitting the data, submitting, managing the jobs and gathering the results.

3. Results

As mentioned in the introduction, we want to generate not only one correlation network, but many, in order to investigate both static and dynamic properties of the time series data. We use sliding windows, evaluating the correlation on an interval T_i of time t, and then sliding that interval by k positions and evaluating it again, producing a series of correlation matrices for each interval T_i. Each of these matrices has one value for the correlation of each pair of channels, and defines different correlation networks that evolve as time goes by. An example data partition of nine intervals for three processors is graphically depicted in Fig. 2.

We can use Adapa to automate this partition scheme, dividing the data and feeding each executable with the values for all channels in its partition. Note that we could also divide the problem in more than one dimension (for example analyzing only certain channels withing the same job) but this might create some repetition in the data among different jobs (the same part of the same channel would have to be on every node that was calculating a correlation with that part of the channel), which would cost time. As long as a single correlation interval is small enough, making it possible to divide the data using each correlation interval as an indivisible unit, this simple scheme of partitioning among one dimension is optimal in what respects to the maximum gain in efficiency. The reason for this is that we can provide equal shares to each processor and minimize overlapping data between them.

[2] Currently the tool is available by special request by sending a mail to the contact author.

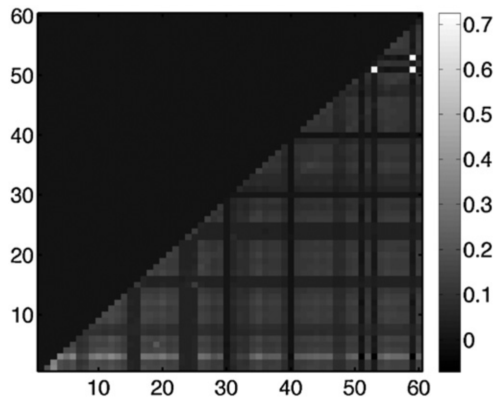

Fig. 3. Resting state cross-correlations across all channels. Channel pairs make up the x- and y-axes, and gray scale color indicates the intensity of correlation between the channels. Since the measure is symmetric between pairs, only half of the data points were calculated and shown (reflecting along the $ch_a = ch_b$ line). In this plot we see that some channels are generally much more correlated with other channels.

The real data used in this paper was generated within the CARMEN project (http://www.carmen.org.uk). CARMEN (Code, Analysis, Repository and Modelling for e-Neuroscience) is an e-science project that seeks to provide an infrastructure in which code, data, etc., can be shared between neuroscience researchers. The particular dataset used was spontaneous firings from a wild-type (WT) mouse retina recorded on a MEA. This is an interesting test-bed for spatio-temporal measures because over the developmental period, the retina changes from a wave-supporting medium (with more short-term connections) to one, which no longer supports waves, but has developed long-term connections within the retina. For analysis purposes, data is not filtered, but the DC offset is removed. Figs. 3 and 4 show preliminary results from cross-correlation analysis across regions; Fig. 3 shows resting state correlation (between successive waves), while Fig. 4 shows dynamical activation (during a wave), highlighting which nodes are highly correlated with which other nodes.

3.1. Type of correlation and data used

In our tests, we experimented with code based on the following correlation formula (which outputs a number between −1 and 1

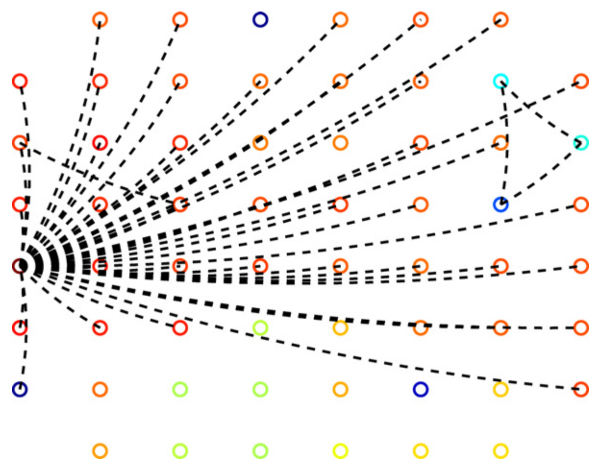

Fig. 4. Dynamic cross-correlations across all channels during a bursting event on the retina. Circles represent the physical electrode arrangement of the MEA (8 × 8 array with four corners missing), and lines between electrodes represent a pairwise correlation above the threshold of 0.3. The color of the electrode indicates the average correlation strength of that channel (to all other channels; i.e. resting state). During this bursting event, connectivity is markedly different from the resting state connectivity.

Table 1
Serial calculation time in seconds for *C* channels and *V* values (factors in parentheses are relative to the scenario with 64 channels (of random data) and 250,000 values).

C	V			
	250,000	500,000	1,000,000	2,000,000
64	7.71 s (=1.0)	14 s (× 1.8)	30 s (× 3.9)	60 s (× 7.8)
128	30 s (× 3.9)	59 s (× 7.7)	124 s (× 16.1)	251 s (× 32.6)
256	122 s (× 15.8)	244 s (× 31.6)	498 s (× 64.6)	1002 s (× 130.0)
512	490 s (× 63.6)	988 s (× 128.1)	1978 s (× 256.5)	3996 s (× 518.3)

indicating the correlation between x and y):

$$\mathrm{Corr}(x,y) = \frac{\sum(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum(x_i-\bar{x})^2}\sqrt{\sum(y_i-\bar{y})^2}} \tag{1}$$

We calculated the correlation in several data files of 25 kHz, 60-channel MEA retina data. We also used random data, in order to study the scalability with an increasing number of channels or other parameters that were not present in the experimental data.

In what follows in this section, C indicates the number of channels of the experiment and V indicates the number of different values in one channel's time series. All random data is uniformly generated with numbers between 0 and 5000.

We will only measure the time it takes to go from the original data file to the correlation matrices, since they implicitly already define the correspondent network, and the real bulk of the work is the all-to-all correlation calculation.

We ran our tool with correlations intervals of size 20,000 (∼0.8 s), with a sliding window of 10,000 (∼0.4 s overlap).

3.2. Evaluation

We evaluated Adapa in three different kinds of environments. The first one was a dedicated Condor cluster (HP ProLiant server) with four quad-core Xeon processors (2.93 Ghz), each one with 8 GB of memory, totaling 32 GB. This cluster has a shared file system (no need to copy files around) and therefore provides a very reliable and self-contained environment for repeated experimentation. Due to other experiments running we were limited to using a maximum of 10 processors at any given time, but those cores were always available, meaning that if we divided the data in 10 or fewer partitions, then the same number of CPUs would be used in parallel at the same time. This dedicated cluster was used for obtaining the results in Sections 3.2.1 and 3.2.2.

The second one was the Newcastle University Condor Pool, which had available more than 200 Linux computers. This is a much more heterogeneous environment (in terms of processor speed, available memory, etc.) and much less resilient (Condor harvests unused computation cycles—if someone uses the computer keyboard, for example, the computation stops) and unreliable (given the dimension, there can be computers with hardware failures). This environment was used for the results in Section 3.2.3.

The third and last environment used was inter-cluster. This is a dedicated cluster with 12 SuperMicro Twinview Servers for a total of 24 nodes. Each node has 2 quad core Xeon 5335 processors and 12 GB of RAM, totaling 192 cores, 288 GB of RAM, and 3.8 TB of disk space, using Infiniband interconnect. This cluster was used for the results in Section 3.2.4.

3.2.1. Serial calculation

In this section we show how our correlation algorithm behaves when being run in serial mode, that is, when no parallelism is explored. This can give us a better feel of how much time it takes to calculate something and if parallelism is really needed. As said before, we used the dedicated cluster for this, but only with one core active at a given time. We used random data (64 channels, chosen to closely mimic the computation load of current MEA analysis needs), in order to produce the desired number of values. Table 1 summarizes the results.

All times are compared to the time taken for 64 channels (of random data) and 250,000 values, which was therefore the only value calculated with more precision. Its final value is the average of 5 runs, rounded to 2 decimal digits. All other values are rounded to seconds.

The table shows that, as expected, the runtime has a behavior asymptotically similar to C^2V. With this in mind we can easily have a rough estimate of how much time a given correlation calculation is going to take in serial on this environment:

$$\mathrm{time}(C,V) = 7.72 \times \frac{V}{250,000} \times \left(\frac{C}{64}\right)^2 \tag{2}$$

For example, with the current usual 60 channels, calculating correlation of 1 h of data at 25 kHz (90,000,000 values) would take 46 min. If we wanted the same calculation for 1000 channels, we would need almost 8 days!

3.2.2. Small dedicated cluster

We ran our algorithm on the dedicated small cluster with a real data file of 60 channels with about 6 min of data (8,735,000 values). We varied the number of partitions, and therefore the number of CPUs used (we always had enough CPUs to assign one per partition). The results we obtained are depicted on Fig. 5. The time spent is the total, including all steps of the process (dividing, submitting, calculating and aggregating). We can also perceive the partial weight of the division step.

We can see that as we use more processors, Adapa takes less time to execute. Despite the time for dividing remaining almost constant (at 3 s), it starts to influence the global time more and more since the total time really becomes small. However, the real bottleneck of

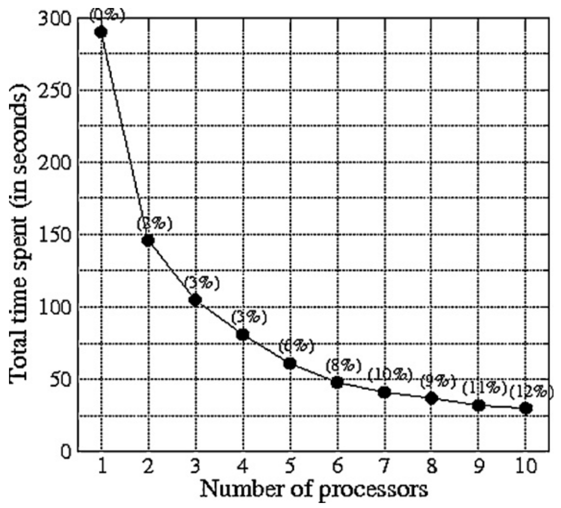

Fig. 5. Time spent in calculating correlations matrices for a real data file. The file contains 60 channels of data, each one with 8,735,000 values. In parenthesis we can see the percentage of time spent in dividing the data (which is done in serial).

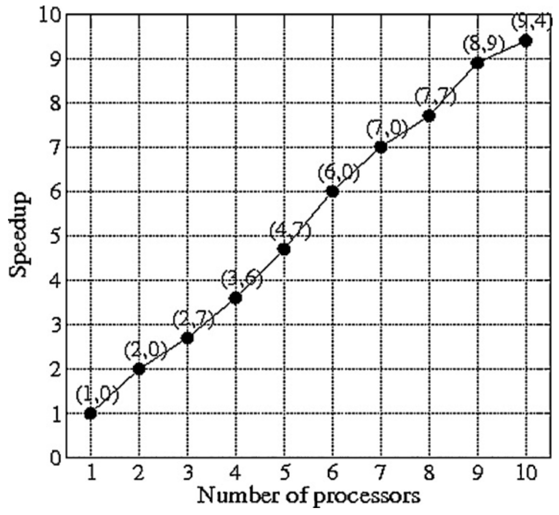

Fig. 6. Speedup when calculating correlation matrices for a real data file. The date is the same as in Fig. 5.

Table 2
Time to calculate random data when we vary the number of values.

$C = 64$		
#V	CPUs	Time spent
2,000,000	1	68 s (=1.0)
4,000,000	2	67 s (× 0.99)
8,000,000	4	70 s (× 1.03)
16,000,000	8	73 s (× 1.07)

the application, as expected, is the correlation computation itself. If we use artificial random data, the results are similar, because, the calculation itself does not depend on the actual values used. It only depends on the quantities assigned to each parameter (number of channels, number of values, size of windows, and size of slide).

Fig. 6 shows the ratio of the time Adapa takes to execute using n processors relative to the time it takes on a single processor. This is called the speedup. We can see that it is almost linear with a slope close to one. This shows that up to this size of the problem, our approach appears to be scalable.

Let us examine the scalability of our application by varying some of the parameters that can influence the time spent on the calculation. We want to support the assumption that the algorithm used is proportional to the number of values by showing that as we double the number of values we need twice as many processors to finish in the same time. Table 2 shows what happens when we fix everything except the number of values V and the number of CPUs, confirming the expected.

We also want to show that as we double the number of channels, we need four times more processors to compute in the same time, which would agree with the notion of time being proportional to the square of the channels. Table 3 shows results from when we fix everything except the number of channels C and the number of CPUs, also confirming what we previewed. Note that we do not use the quadruple number of processors in each following iteration since we had a limited number of processors available.

All of this shows that Adapa seems to scale well and as expected (given that the complexity beneath the calculation) grows in a way asymptotically similar to C^2V.

Table 3
Time to calculate random data when we vary the number of channels.

$V = 2,000,000$		
#C	CPUs	Time spent
64	1	68 s (=1.0)
128	2	131 s (× 1.93)
256	4	254 s (× 3.74)
512	8	518 s (× 7.62)

3.2.3. Newcastle University Condor pool

In the Newcastle Condor pool, with dynamic, competitive and heterogeneous environment, the best option seems to be to divide the data as much as possible, as long as the different partitions take some time (more than a few seconds). We lose some time when starting the jobs and they have a small fraction of shared data between them, but we gain more robustness to unreliability (because stopping a job, will cost us less) and heterogeneity (because faster processors can process multiple chunks of data – jobs – during the time another processor calculates only one). In this case, we also need to transfer files, since there is no global shared file-system.

The time it takes to run a complete calculation is very difficult to predict and it depends on the current system conditions and network topology. What matters most is that, in practice, the resource can be used to effectively cut the time needed to calculate the correlation networks. Adapa was effectively used to submit jobs to the pool. The current experiment only uses Linux executables but if one had a different operating system available, we would also be able to submit the respective binaries.

We first used the same real data file described in Section 3.2.2. We divided it into 50 small partitions and the whole computation took less than 3 min to execute and produced a correct and valid output, despite the complex environment. The maximum amount of computers used at the same time was 13, but this was only because the Condor system did not assign us more.

We also ran an experiment with 16 different real data files, also with retina data (25 kHz, 60 channel). We divided each one in 5 chunks and submitted all 80 partitions. This would take more than 8 h on a single fast processor of the dedicated cluster we used before (2.93 GHz) but on the Condor Pool it took only 25 min (using at most 26 computers at the same time). Again, the results were correctly calculated, with no error reported, with Condor providing the quality-of-service and guaranteeing that when a job was abnormally terminated, it would be respawn (that is, resubmitted to another processor in the pool).

A better study of these parallel performance results is out of the scope of this paper. We mostly wanted to show that in a simple way we can effectively take advantage of commonly existing computing resources that are otherwise not being used. Adapa effectively leverages the access to a complex high performance system.

3.2.4. Inter-cluster

We wanted to assess the potential of Adapa in a larger scale and in a different batch system. The inter-cluster provided us with that opportunity. Inter-cluster uses the torque/PBS submissions system, which was an opportunity to prove that Adapa can be extended in a simple way to other batch submissions systems. By using again a command line interface, the tool can communicate with it in a similar fashion to Condor. This of course required some programming effort from our side, but it is only made once and then any scientist can use any system with that particular submissions system. In practical terms, basically we have to understand the syntax of Torque and code a single module implementing all primitive functions, such as sending a job or checking its status. After doing that, from the point of view of the user, it is exactly the same as using Condor. Obviously, a parameter must be set indicating that we will be using torque.

Table 4 shows the results of running the correlation calculation for a piece of random data with 1,290,000 values and 1000 channels

Table 4
Time to calculate a random data file in inter-cluster (factors in parentheses are the speedup relative to the scenario with one CPU).

#CPUs	Total time	Time in calculation	% time in division
1	15089.1 s (=1.0)	15089.1 s (=1.0)	0.00%
2	7052.8 s (× 2.14)	6905 s (× 2.19)	2.10%
4	3787.3 s (× 3.98)	3637 s (× 4.50)	3.97%
8	2277.1 s (× 6.63)	2127 s (× 7.09)	6.59%
16	1430.5 s (× 10.55)	1275 s (× 11.83)	10.87%
32	1105.2 s (× 13.65)	947 s (× 15.93)	14.31%
64	1024.9 s (× 14.71)	867 s (× 167.50)	15.41%

up to 64 processors. Note that the size of each correlation interval (20,000) and the size of the sliding window (10,000) is still the same as in the other experiments. Therefore, 1,290,000 is equally dividable by any number power of two, precisely the number of processors that we are going to test.

We start with a calculation that takes more than 5 h in serial and we end up spending less than 20 min with 64 processors.

However, the scalability is far from optimal. For once, it is limited by the division of the files, which is made in serial and therefore takes an almost constant time (varying between 147 and 157 s). More than that, in this specific case, we made the division on the cluster itself and it takes more time than on a commodity computer since the file system is more complex and network shared. One way to avoid this division bottleneck would be for the submitted binaries to directly access the original data file and merely take in consideration which part of the data it should access. If the file system used is capable of supporting multiple access, this could provide a substantial improvement in the speedup.

Note also the speedup if we take only into consideration the calculation step. It is lower than expected (since this part is completely parallel, the optimum would be linear speedup). This can be due to many factors, like the network congestion when multiple jobs are accessing files on the shared file system. Note also that no notion of data locality is used, since all executables try to access the same file system. Further, we should also note that contrary to Condor, this particular implementation of torque did not provide the built-in capability of array submission, that is, the capacity to understand that several copies of the same executable are actually from the same job and being executed at the same time. This can lead to a bad use of the cores, in the sense that it is not guaranteed that different jobs on the same node are really using different cores. However, other systems could provide this array functionality and Adapa would use that if possible, leading to potentially better results. As a side effect of all this, certain nodes are finishing their calculations earlier than others and therefore potential computation power is being wasted.

The bottom note is that again Adapa was used to obtain correct results, this time on a different environment and with a much larger scale.

4. Discussion

In this paper we presented a method to calculate correlation networks in parallel. By doing that, we also presented a tool and the architecture to aid in automating data distribution and job submission for applications that exhibit parallelism at data level. The tool also allows the usage of different programming languages for the implementation of the algorithms.

We implemented an initial functional prototype and proved that it can be used to achieve higher throughput in a single cluster, obtaining an almost linear speedup for an example real application. We also used it in a real Condor pool and paved the way for its use on a more heterogeneous environment.

During the experimentation we discovered that our application has the potential for being scalable to a large number of processors. However, there is a limitation in that we must choose a problem big enough in terms of real calculation to minimize the influence of the division, which is made in serial, exploiting no parallelism at all. Such a problem would be motif discovery (Milo et al., 2002; Sporns and Kötter, 2004) in correlation networks where the bottleneck is not only given by the number of nodes in the network and the size of the motifs but also by how many different correlation networks can be generated within a recording session.

There are also several possible improvements on the software development side. First, we need to have access to a real life more up-to-date Condor pool to better test Adapa's behavior. This will be used to test additional features such as data compression before sending data over the network. We would also like to test other applications with our platform, particularly written in other programming languages. Second, we also think it would be beneficial to skip the division phase and directly access the data in shared file systems. Such a procedure could take advantage of dedicated clusters and provide better results using less time. Third, we want to compare our approach in terms of results, to a more specialized solution like Hadoop (Bialecki et al., 2009). This could be done by letting Adapa interface between the user and the Hadoop platform. Finally, we plan to experiment in how Adapa could be used for real-time analysis of a stream of data (currently it is more suited to offline analysis of previously obtained data).

In conclusion, this work shows that our tool can be used for parallel processing of neuroscience multi-channel data across a range of operating and grid computing systems. By providing a simple and extendable tool, we hope to reduce the entry barriers for using parallel computing to analyze large-scale neuroscience data sets.

Acknowledgements

We thank Enrico Pontelli for the use of inter-cluster in the New Mexico State University. Within CARMEN, we particularly thank Evelyne Sernagor's lab for providing the experimental data with which this code was tested. This study was financially supported by FCT (SFRH/BD/19753/2004), the Royal Society (RG/2006/R2), and EPSRC (EP/E002331/1).

References

Achard S, Bullmore E. Efficiency and cost of economical brain functional networks. PLoS Comput Biol 2007;3(2):e17.
Albert R, Barabási AL. Statistical mechanics of complex networks. Rev Mod Phys 2002;74(1):47–97.
Bialecki A, Cafarella M, Cutting D, O'Malley O. Hadoop: a framework for running applications on large clusters built of commodity hardware. (August 2009) http://hadoop.apache.org/.
Chapman C, Goonatilake C, Emmerich W, Farrellee M, Tannenbaum T, Livny M, et al. Condor BirdBath: Web service interfaces to Condor. In: Cox SJ, Walker DW, editors. Proceedings of the UK e-Science All Hands Meeting 2005. Nottingham, UK; 2005: 737–44.
Costa LDF, Rodrigues FA, Travieso G, Villas Boas PR. Characterization of complex networks: a survey of measurements. Adv Phys 2007;56:167–242.
Eckersley P, Egan GF, Amari S, Beltrame F, Bennett R, Bjaalie JG, et al. Neuroscience data and tool sharing: a legal and policy framework for neuroinformatics. Neuroinformatics 2003;1:149–65.
Eglen SJ, Adams C, Echtermeyer C, Kaiser M, Simonotto J, Sernagor E. The carmen project: Large-scale analysis of spontaneous retinal activity during development. In: European retina meeting; 2007.
GAHP, Grid ASCII Helper Protocol. (August 2009) http://www.cs.wisc.edu/condor/gahp/.
Gentzsch W. Sun grid engine: towards creating a compute power grid. In: Buyya R, Mohay G, Roe P, editors. Proceedings of the first IEEE/ACM international symposium on cluster computing and the grid. USA: IEEE Press; 2001. p. 35–6.
Ince RA, Petersen RS, Swan DC, Panzeri S. Python for information theoretic analysis of neural data. Front Neuroinform 2009;3:4.
Jurica P, van Leeuwen C. OMPC: an Open-Source MATLAB-to-Python Compiler. Front Neuroinform 2009;3:5.
Martino RL, Johnson CA, Suh EB, Trus BL, Yap TK. Parallel computing in biomedical research. Science 1994;265:902–8.

Meier R, Egert U, Aertsen A, Nawrot MP. FIND—a unified framework for neural data analysis. Neural Networks 2008;21:1085–93.

Micheloyannis S, Pachou E, Stam C, Breakspear M, Bitsios P, Vourkas M, et al. Small-world networks and disturbed functional connectivity in schizophrenia. Schizophr Res 2006;87:60–6.

Milo R, Shen-Orr S, Itzkovitz S, Kashtan N, Chklovskii D, Alon U. Network motifs: simple building blocks of complex networks. Science 2002;298(5594):824–7.

Mitra S, Kothari SC, Cho J, Krishnaswarmy A. Paragent: a domain-specific semi-automatic parallelization tool. In: Valero M, Prasanna VK, Vajapeyam S, editors. HiPC, vol. 1970. Springer; 2000. p. 141–8.

Oppenheim MI, Factor M, Sittig DF. BIO-SPEAD: a parallel computing environment to accelerate development of biologic signal processing algorithms. Comput Methods Programs Biomed 1992;37:137–47.

Pancake CM. Usability issues in developing tools for the grid—and how visual representations can help. Parallel Process Lett 2003;13(2):189–206.

PBS, OpenPBS at PBS gridworks. (August 2009) http://www.openpbs.org.

Ponten S, Bartolomei F, Stam C. Small-world networks and epilepsy: graph theoretical analysis of intracerebrally recorded mesial temporal lobe seizures. Clin Neurophysiol 2007;118:918–27.

Ribeiro P, Simonotto J, Kaiser M, Silva F. Adapa download site. (September 2009) http://www.dcc.fc.up.pt/adapa/.

Salvador R, Suckling J, Coleman MR, Pickard JD, Menon D, Bullmore E. Neurophysiological architecture of functional magnetic resonance images of human brain. Cereb Cortex 2005;15(9):1332–42.

Smith L, Austin J, Baker S, Borisyuk R, Eglen S, Feng J, et al. The CARMEN e-Science pilot project: Neuroinformatics work packages. In: Cox, S.J., editor. Proceedings of the UK e-Science All Hands Meeting 2007. Nottingham, UK; 2007: 591–98.

Spacek M, Blanche T, Swindale N. Python for large-scale electrophysiology. Front Neuroinform 2008;2:9.

Sporns O, Chialvo DR, Kaiser M, Hilgetag CC. Organization, development and function of complex brain networks. Trends Cogn Sci 2004;8:418–25.

Sporns O, Kötter R. Motifs in brain networks. PLoS Biol 2004;2(11).

Stam C, Jones B, Nolte G, Breakspear M, Scheltens P. Small-world networks and functional connectivity in Alzheimer's disease. Cereb Cortex 2007;17:92–9.

Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. Concurrency-Pract Ex 2005;17(2–4):323–56.

Torque, Torque Resource Manager at Cluster Resources, Inc. (September 2009) http://www.clusterresources.com/products/torque-resource-manager.php.

Watson P, Jackson T, Pitsilis G, Gibson F, Austin J, Fletcher M, et al. The CARMEN neuroscience server. In Cox SJ, editor. Proceedings of the UK e-Science All Hands Meeting 2007. Nottingham, UK; 2007: 135–41.