Journal of Neuroscience Methods 257 (2016) 7–16

Contents lists available at ScienceDirect

Journal of Neuroscience Methods

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

Computational neuroscience

mpdcm: A toolbox for massively parallel dynamic causal modeling

Eduardo A. Aponte[a,*], Sudhir Raman[a], Biswa Sengupta[b], Will D. Penny[b], Klaas E. Stephan[a,b,c], Jakob Heinzle[a]

[a] Translational Neuromodeling Unit (TNU), Institute for Biomedical Engineering, University of Zurich and Swiss Federal Institute of Technology (ETH), 8032 Zurich, Switzerland
[b] Wellcome Trust Centre for Neuroimaging, University College London, 12 Queen Square, London WC1N 3BG, UK
[c] Max Planck Institute for Metabolism Research, 50931 Cologne, Germany

HIGHLIGHTS

- "mpdcm" is a toolbox for fast simulation of Dynamic causal models for fMRI on GPUs.
- Parallelization reduces computation time by up to two orders of magnitude.
- This enables the use of sampling algorithms for Bayesian inference.
- The mpdcm toolbox is openly available under the GPLv3 license.

ARTICLE INFO

Article history:
Received 13 May 2015
Received in revised form 15 August 2015
Accepted 8 September 2015
Available online 16 September 2015

Keywords:
Dynamic causal modeling
GPU
Markov chain Monte Carlo
Thermodynamic integration
Parallel tempering
Model inversion
Model evidence
Bayesian model comparison

ABSTRACT

Background: Dynamic causal modeling (DCM) for fMRI is an established method for Bayesian system identification and inference on effective brain connectivity. DCM relies on a biophysical model that links hidden neuronal activity to measurable BOLD signals. Currently, biophysical simulations from DCM constitute a serious computational hindrance. Here, we present *Massively Parallel Dynamic Causal Modeling* (**mpdcm**), a toolbox designed to address this bottleneck.
New method: **mpdcm** delegates the generation of simulations from DCM's biophysical model to graphical processing units (GPUs). Simulations are generated in parallel by implementing a low storage explicit Runge–Kutta's scheme on a GPU architecture. **mpdcm** is publicly available under the GPLv3 license.
Results: We found that **mpdcm** efficiently generates large number of simulations without compromising their accuracy. As applications of **mpdcm**, we suggest two computationally expensive sampling algorithms: thermodynamic integration and parallel tempering.
Comparison with existing method(s): **mpdcm** is up to two orders of magnitude more efficient than the standard implementation in the software package SPM. Parallel tempering increases the mixing properties of the traditional Metropolis–Hastings algorithm at low computational cost given efficient, parallel simulations of a model.
Conclusions: Future applications of DCM will likely require increasingly large computational resources, for example, when the likelihood landscape of a model is multimodal, or when implementing sampling methods for multi-subject analysis. Due to the wide availability of GPUs, algorithmic advances can be readily available in the absence of access to large computer grids, or when there is a lack of expertise to implement algorithms in such grids.

1. Introduction

Dynamic causal modeling (DCM) (Friston et al., 2003) is a widely used Bayesian framework for inference on effective connectivity from neurophysiological data. When applied to fMRI, it is a hierarchical generative model that integrates an explicit model of neuronal population interactions with a biophysical model of

* Corresponding author. Tel.: +41 44 634 91 12; fax: +41 44 634 9131.
E-mail address: aponte@biomed.ee.ethz.ch (E.A. Aponte).

http://dx.doi.org/10.1016/j.jneumeth.2015.09.009
0165-0270/

the regional blood oxygen level dependent (BOLD) signals caused by those interactions. Currently, parameter estimation in DCM rests on a variational Bayesian scheme which maximises negative free energy (Friston et al., 2003, 2007). Alternative inference techniques have been proposed, for example, Markov chain Monte Carlo (MCMC) sampling (Chumbley et al., 2007; Raman et al. unpublished results, see also Sengupta et al., 2015 for an MCMC approach to DCM for electrophysiological data). However, variational Bayes has not yet been replaced as the method of choice for inference in DCM. One reason for this is computational efficiency: sampling methods are limited by the high computational cost of evaluating DCM's likelihood function, because this requires integrating the neuronal as well as the hemodynamic model equations in DCM. (For simplicity, in the following we use the term "simulations" to refer to integration of the model's state equations.)

In this paper, we introduce *Massively Parallel Dynamic Causal Modeling*, **mpdcm**, a toolbox designed to overcome some of the computational limitations of DCM. **mpdcm** delegates the simulations of DCM's biophysical model to graphical processing units (GPUs). The goal of our toolbox is to facilitate the implementation of algorithms for statistical inference in DCM that require large numbers of simulations, as in the case of multi-subject hierarchical models (Raman et al. unpublished results) or sampling methods. **mpdcm** is mostly written in the C programming language and is accompanied by a MATLAB interface compatible with Statistical Parametric Mapping (SPM). It is available under the GPL license as part of the open source TAPAS software at www.translationalneuromodeling.org/software.

There is a growing interest in using GPUs in the context of imaging (Shi et al., 2012; Eklund et al., 2013) and particularly in the field of MRI. For example, Hernandez et al. (2013) used a massively parallel algorithm to accelerate a sampling method for diffusion weighted imaging. In the case of fMRI, mature toolboxes covering several aspects of the analysis pipeline, including coregistration and statistical inference, are available (Eklund et al., 2014). Ferreira da Silva (2011) used GPUs to apply Bayesian inference to BOLD time series modeling using sampling methods. Recently, Jing et al. (2015) proposed to accelerate group ICA by using GPUs, a method integrating a toolbox originally developed for EEG (Raimondo et al., 2012). In the context of DCM, Wang et al. (2013) presented a massively parallel implementation of DCM for event-related potentials. This implementation aimed mostly at speeding up the variational EM algorithm for inference in DCM for EEG. Here, we propose a more general approach in the context of DCM for fMRI that can not only increase the efficiency of the variational EM algorithm, but also the efficiency of, for example, sampling methods. An introductory account of parallel programming can be found in Suchard et al. (2010).

In the following, we first briefly introduce DCM and present our toolbox. We then compare the accuracy and performance of **mpdcm** to the standard implementation in SPM. We also showcase the implementation of path sampling and parallel tempering, two computationally expensive sampling algorithms that can be easily optimized with **mpdcm**. Finally, we discuss future applications, extensions, and limitations.

2. Methods

2.1. Dynamic causal modeling

Dynamic causal models describe continuous time interactions between nodes (neuronal populations) in a pre-specified neural network. The time course of neuronal activity is modeled bythe bilinear differential equation

$$\dot{x} = Ax + \sum_{n=1}^{N} u_i, B_i x + C_u, \tag{1}$$

where x is a vector of neuronal states, u is an N dimensional vector of inputs to the network, and A, B, and C are matrices that represent the connection strengths between nodes and the influence of external inputs on nodes and connections. The evolution of x therefore depends on both the interactions between nodes and experimental inputs. In order to predict measurable signals, DCM for fMRI also invokes a biophysical model, which consists of weakly nonlinear differential equations that link neuronal activity to BOLD signals, including an extended Balloon model (Buxton et al., 1998; Friston et al., 2003; Stephan et al., 2007). Briefly, the BOLD signal is modeled as a nonlinear function of the venous compartment volume and the deoxyhemoglobin content (Stephan et al., 2007). The venous compartment is assumed to behave as a balloon, where the total blood volume is the difference of the blood inflow and outflow. The deoxyhemoglobin content is simply the difference of (1) the product of the blood inflow and oxygen extraction rate and (2) the outflow rate times the deoxyhemoglobin concentration. Importantly, blood inflow into the venous compartment is triggered by putative brain activity x. For a detailed treatment see Stephan et al. (2007). Because the Balloon model comprises nonlinear differential equations that cannot be solved analytically, numerical integration is necessary to generate predictions from the model. To simplify notation from here on, we refer to all states (neuronal or hemodynamic) as x and the state equations as f, such that:

$$\dot{x} = f\left(x, u, \theta\right), \tag{2}$$

where u are inputs, and θ are subject specific parameters, including matrices A, B and C, as well as several hemodynamic parameters.

States x are linked to observable quantities via a forward model that we denote by the function g:

$$\hat{y} = g\left(x, u, \theta\right), \tag{3}$$

where $\hat{y}$ is the predicted BOLD signal.

DCM can be used in two different ways: identification of system structure and parameter estimation. The former requires model comparison, i.e., determining which model, from a predefined set of alternatives (model space), best explains a set of empirical observations. Typical differences between models include the absence or presence of certain connections or modulations thereof (entries in matrices A or B). The latter consists in determining the posterior distribution of the parameters θ given experimental observations and a model (Penny et al., 2010). Summary statistics of these distributions can be used, for example, as dependent variables in statistical tests, which can indicate the effect of a treatment or condition on effective connectivity.

Model comparison (Penny et al., 2004) and parameter estimation (Friston et al., 2003) rest on the generative nature of DCM. A generative model requires two things: first, a likelihood function that defines the probability of observed data given a prediction or simulation of the model. Model predictions are fully specified by the parameters θ, experimental input u, the neuronal state equation $f(x, u, \theta)$, and the forward model $g(x, u, \theta)$. Second, a generative model requires a prior distribution of the parameters θ, i.e., the probability of θ before any observation. The relationship between these elements is determined by Bayes' formula:

$$p\left(\theta \left| y, m\right.\right) = \frac{p\left(y \left| \theta, m\right.\right) p\left(\theta \left| m\right.\right)}{p\left(y \left| m\right.\right)}, \tag{4}$$

where y is a set of experimental observations, θ are the parameters, and m is the model being evaluated. In Bayesian statistics,

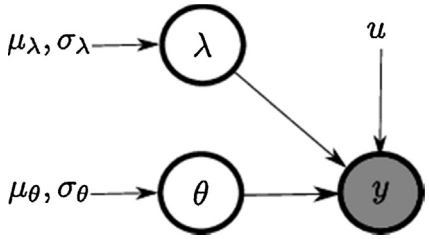

Fig. 1. Graphical representation of the generative model underlying DCM. All distributions are Gaussian. Experimental data y is modeled as being conditionally dependent on the model parameters θ and a region-wise noise scaling parameter λ.

parameter estimation (model inversion) is equivalent to computing the posterior probability $p\left(\theta|y, m\right)$. Identification of system structure can be achieved by comparing different models on the basis of their marginal likelihood or model evidence $p(y|m)$ (Kass and Raftery, 1995; Penny et al., 2004). This quantity is the probability of a set of observations y given a model m after marginalizing over parameters θ. Comparing models via their marginal likelihood is an alternative or complementary approach to cross-validation and other methods routinely used in frequentist statistics for model comparison (Kass and Raftery, 1995; Gelman et al., 2003; MacKay, 2003)

DCM is defined by a Gaussian noise model and Gaussian priors (Friston et al., 2003). Assuming that there are $r = 1, \ldots, R$ nodes or brain regions, the model is specified by the joint probability

$$p\left(y_1, \ldots, y_R \left|\theta, \lambda_1, \ldots, \lambda_R, m\right.\right) = p\left(\theta|m\right) \prod_{r=1}^{R} p\left(y_r \left|\theta, \lambda_r, m\right.\right) p\left(\lambda_r|m\right), \quad (5)$$

$$p\left(y_r \left|\theta, \lambda_r, m\right.\right) = N\left(y_r|g\left(x_r, u, \theta\right), \lambda_r^{-1} I\right), \quad (6)$$

$$p\left(\theta|m\right) = N\left(\theta \left|\mu_\theta, \sigma_\theta^2 I\right.\right), \quad (7)$$

$$p\left(\ln \lambda_r|m\right) = N\left(\ln \lambda_r \left|\mu_\lambda, \sigma_\lambda^2\right.\right), \quad (8)$$

where N denotes the Gaussian probability density function, y_r corresponds to the time series extracted from brain region r, λ_r corresponds to a region-specific scaling factor of the covariance of the noise of the observed BOLD signal, and I is an identity matrix (assuming temporal auto-correlation in the BOLD signals has been removed by whitening). A graphical representation of the model is presented in Fig. 1.

In summary, classical DCM for fMRI assumes a bilinear model of the interactions between brain regions and a nonlinear model of the ensuing BOLD signal. Simulations of the model are used to construct a likelihood function by assuming that the observed BOLD signal is Gaussian distributed around the predicted signal. Given the nonlinear model of the hemodynamics that cause the BOLD signal, predicting data points requires numerical integration of the differential Eq. (2).

The main computational hurdle for DCM is that it requires the simulation of predicted BOLD signals. The goal of **mpdcm** is to accelerate the numerical integration of the biophysical model of DCM by exploiting the computational power of GPUs. Here, we optimized only the generation of biophysical simulations, because the computational costs of DCM are mostly incurred by the simulation step. This implies that **mpdcm** can be used to optimize any inference algorithm that relies on a model with the same likelihood function, independently of any other aspects of the model. In the next section, we briefly present the implementation details of our toolbox.

2.2. Massively parallel simulation of DCMs

The temporal integration of DCM's biophysical model on massively parallel architectures is accompanied by three main challenges: First, the problem is iterative in nature. This property forces a serial implementation of the time updates and imposes a ceiling to the performance gains achievable. Thus, part of our goal was to exploit parallelizations on each update step to minimize the serial component of the algorithm. Second, the integration of DCM's dynamic system cannot be rendered into the framework of single instruction/multiple data paradigm, where GPU architectures excel, because each state of the model evolves according to different dynamics. Third, connectivity between areas and the dynamics of the states require high levels of communication between threads and thus increase the effect of memory latencies.

We implemented two integration schemes in **mpdcm**. First, we implemented an integrator based on Euler's method (Butcher, 2008), which approximates x at time $t+h$ through the equation:

$$x_{t+h} = x_t + hf\left(x_t, u_t, \theta\right). \quad (9)$$

Euler's integration scheme does not require the storage of any intermediate result between iterations and hence has low memory requirements. Although memory is nowadays often not a major concern in CPU-based computations, memory access and size are usually the main bottlenecks for GPU-based computations. The advantages of Euler's integration come at the price of reduced accuracy and, more importantly, possible accumulation of errors. In order to reduce these inaccuracies, we also implemented a fixed step sized Runge–Kutta scheme of fourth order (Butcher, 2008). This standard technique uses a linear combination of different, iterative evaluations of f to approximate the dynamics of x. Runga–Kutta's method requires four times more floating point operations that cannot be parallelized and, more importantly, twice as much memory for the storage of intermediate results. In order to reduce the memory needed to store intermediate steps, we implemented a modified version of Runge–Kutta's method (Blum, 1962). This modification takes advantage of linear dependencies between the coefficients of f to reduce the number of required registers for each state variable from four to three registers. Both integration methods are compared in Section 3.

In our GPU implementation, we exploited three sources of parallelism. First, simulations for systems with different sets of parameters θ or inputs u can be computed in parallel. As it will become clear in the next section, several applications require simulations from the same model with different parametrizations. Second, the update of each node of the network can be performed in parallel. Finally, because f is a multivariate function, its evaluation in each dimension can be parallelized. Similarly, we parallelized the evaluation of the forward model g for each region.

An important consideration in our implementation is memory access. In contrast to CPUs, GPUs have only a very limited amount of fast memory that can be shared between threads in a block. Currently, a single multiprocessor has by default access to 48 kb of high latency shared memory, and access to a high latency global memory of several gigabytes. In **mpdcm**, the experimental inputs u are loaded to global memory and kept there, while model parameters are stored in shared memory. States are stored in shared memory, and predicted output is stored in global memory and transferred to the CPU memory after execution. This implies that each time step requires the access to slow global memory.

In Algorithm 1 we present pseudo code of our implementation of Euler's integration method. Several details have been

left out in order to clarify the main aspect of the implementation. Euler's method can be easily extended to accommodate the Runge Kutta's fourth order method. The implementation of the Runge Kutta's algorithm is described in the supplementary materials.

Algorithm 1:
Requires: *itheta*—array of pointers to model parameters
Requires: *ntheta*—number of elements of *itheta*.
Requires: *iu*—array of experimental inputs.
Requires: *iy*—arrays of pointers to predicted signals.
Requires: *nt*—total number of integration steps.
Requires: *dt*—1 over sampling rate (integer)
Requires: *h*—time constant.
Requires: *nregions*—number of regions.
Requires: *NUM_SM_PROCESSORS*—number of streaming multiprocessors.
Requires: *MAX_THREADS*—maximal number of threads supported by each multiprocessor.
Requires: *MAX_NUM_NODES*—maximal network size supported.
1: Allocate shared float array *buf*0 of size *MAX_NUM_NODES* * 5.
2: Allocate shared float array *buf*1 of size *MAX_NUM_NODES* * 5.
3: Allocate local pointer to model parameters *theta*.
4: Allocate local pointer to float array *y*.
5: Allocate local pointer to float array u.
6: *nblocks* = *NUM_SM_PROCESSORS* * (*MAX_THREADS*/(*MAX_NUM_NODES* * 5))
7: Set thread block size to (*MAX_THREADS*, 5).
8: Set grid size to (*nblocks*, 1).
9: $l = blockId.x * (MAX\_NUM\_NODES/nregions) + (threadId.x/nregions)$
10: **while** $l <$ *ntheta* in parallel **do**
11: $inrange = threadId.y < nregions * (MAX\_NUM\_THREADS/nregions)$
12: $nlow = nregions * (threadId.y/nregions)$
13: $nhigh = nlow + nregions$
14: $theta = itheta[l]$
15: $u = iu[l]$
16: $y = iy[l]$
17: **for** $t = 0$ **to** $nt - 1$ **do**
18: **for** $i = 0$ **to** *MAX_NUM_NODES* − 1 in parallel **do**
19: **for** $j = 0$ **to** 4 in parallel **do**
20: **if** *inrange* **then**
21: $buf1[i,j] = buf0[i,j] + h * f_j(buf0[nlow : nhigh, 0:4], theta, u[nt], i)$
22: **end if**
23: **end for**
24: synchronize threads
25: **if** $nt\,\%\,dt == 0$ **and** *inrange* **and** $threadId.y == 0$ **then**
26: $y[nt/nd, threadId.x\%nr] = g(theta, buf0[i, 0:4])$
27: **end if**
28: **end for**
29: Swap *buf*0 and *buf*1
30: **end for**
31: $l = l + nblocks * (MAX\_NUM\_NODES/nregions)$
32: **end while**

We exploited three levels of parallelization. First, several simulations can be performed in parallel (line 10). Second, the state equations of each node are computed in parallel (line 18–19). Crucially, by setting *MAX_NUM_NODES* to a multiple of the minimal warp size, different thread warps are assigned to the evaluation of different state equations (line 19). Importantly, to maximize occupancy, the number of blocks is based on the maximal number of possible threads that can be hold by each streaming multiprocessor (line 6). Further implementational details are omitted here for clarity.

2.3. Comparing simulation accuracy

We verified the accuracy of Euler's and Runge–Kutta's method by comparing simulations from **mpdcm** against simulations from SPM's function *spm_int_J.m*, which was introduced for nonlinear DCMs (Stephan et al., 2008). All results are based on SPM8, revision 5236. The implementation in *spm_int_J.m* uses a local linearization of the dynamics of x (Ozaki, 1992). The update of x is given by

$$x_{t+h} = x_t + J^{-1}\left(e^{hJ} - I\right) f\left(x, u, \theta\right), \tag{10}$$

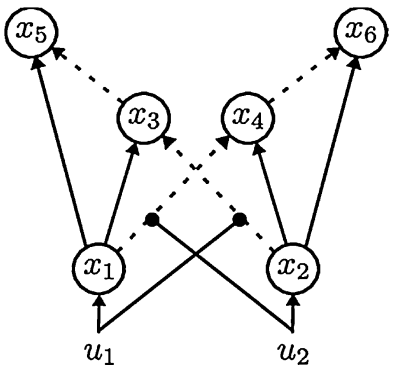

Fig. 2. Exemplary DCM network. Solid arrows represent positively signed (excitatory) connections. Broken arrows correspond to negatively signed (inhibitory) connections. Self-inhibitory connections are not displayed. Inputs u_1 and u_2 activate directly only nodes 1 and 2, and modulate the connections between nodes 1 and 4 and between nodes 2 and 3.

where J is the Jacobian of f with respect to x, and I is the identity matrix. Although this scheme is a highly accurate method, it requires the computation of the Jacobian of f, one evaluation of f, and one matrix exponentiation per update.

To evaluate the three integration schemes, five sets of model parameters of a six node network were drawn randomly. Fig. 2 displays the structure of the network. We assumed a TR of 2.0 s. The inputs u had a sampling rate of 8.0 Hz and consisted of regular box car functions with a width of 20.0 s. Box car functions are commonly used to model DCM inputs as they represent categorical experimental conditions. The step size h used for all simulations was 0.125 s, therefore matching with the sampling rate of u. A total of 512 scans were simulated.

2.4. Comparing simulation performance

We compared the performance of **mpdcm** against the default integrator in SPM used for inference. This method is implemented in *spm_int.m*. It relies on a linear approximation of f

$$\dot{x} = f\left(x, u, \theta\right) \approx J(u)x. \tag{11}$$

Using the same numerical integration scheme as in Eq. (10), which effectively corresponds to the analytical solution assuming linearity of f with respect to x, the iterative updates are given by

$$x_{t+h} = e^{hJ(u_t)}x_t. \tag{12}$$

Because $J(u)$ needs to be recomputed only when u changes, increasing the sampling rate of u when u is composed of only step functions does not change the results nor the number of computations necessary to generate simulations (Friston, 2002). This method is a compromise between computational time and accuracy, as it approximates f with a linear system, but preserves the nonlinearities in g.

To further evaluate the performance gains provided by **mpdcm**, we also implemented a highly optimized, multithreaded *C* version of Euler's method and compared it against **mpdcm**. This function is also available as part of our toolbox. We use *openmp* to multithread our implementation at the level of simultaneous simulations. Although a more fine-grained parallelization is possible, for a large number of simulations this simple scheme warrants constant high occupancy, as no synchronization is required. Moreover, a more fine-grained parallelization would also come at the cost of increased latency in the initialization of parallel threads, a problem not found in GPU units (Suchard et al., 2010).

We compared the differences in timing of a single simulation and then proceeded to test the gains in performance when simulating several models in parallel. Because SPM's method lacks

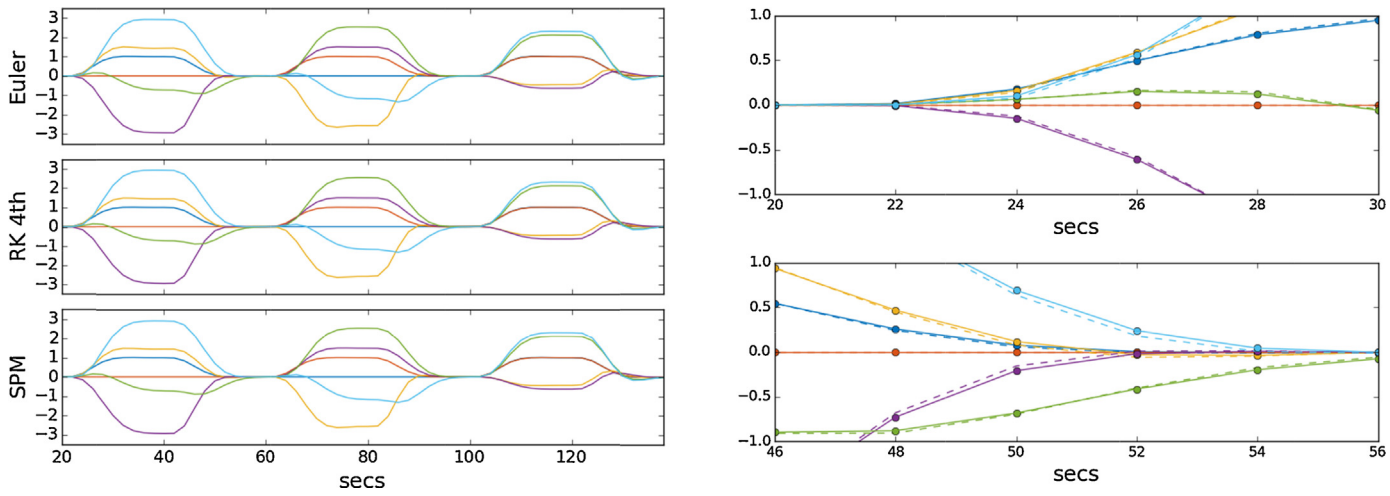

Fig. 3. Left: simulated BOLD signal using SPM, Euler and Runge–Kutta fourth order integration methods. Right: The same data zoomed-in in two different intervals. Solid lines correspond to SPM, broken lines to Euler's method, and circles to Runge–Kutta's method. Differences are only apparent during transients.

any optimization for parallel simulations, its computational time of multiple simulations is simply additive. Thus, we did not apply SPM's method to more than one simulation. The experimental input u used for these simulations had the same parameters as in the previous section.

2.5. Environment

Currently, **mpdcm** targets *Nvidia* graphical processing units and is written mostly in the *C* programming language using the Compute Unified Device Architecture (CUDA) API. *Nvidia* GPUS are off-the-shelf chip sets that make massively parallel architectures affordable for standard desktop computers. **mpdcm** has no dependencies on specialized libraries provided by the manufacturer except for the standard *C* library and the *math* library.

Simulations were performed on a dedicated machine with a twelve core AMD Opteron 6174 processor, a professional *Nvidia* Tesla M2050, Linux CentOS 6.5, and MATLAB 8.1. **mpdcm** was compiled with gcc 4.4.7. The Tesla M2050 has a single precision arithmetic peak performance of 1030.6 giga floating point operations per seconds (GFLOPS), and 512.2 GFLOPS for double precision arithmetic. All GPU and CPU simulations were performed on this machine if not indicated otherwise. In order to investigate the differences in performance that single and double floating point representation could produce, some of the simulations were performed on a machine equipped with an Intel i7 4770k, a *Nvidia* Geforce GTX 760, Linux Ubuntu 13.10, gcc 4.8.1 and MATLAB 8.1. The peak performance of single precision arithmetic of the GTX 760 is 2257.0 GFLOPS and only 94 GFLOPS of double precision arithmetic. It is important to notice that our goal was not to benchmark the two GPUs, but to evaluate the performance change produced by different ratios of single to double precision peak FLOPS. All other simulations were performed in single precision, unless otherwise stated. In the remainder of this paper, we first compare accuracy and performance gains in the simulation of BOLD data in DCM and then explore the application of our toolbox for model comparison.

3. Results

3.1. Accuracy

Fig. 3 shows a fragment of one simulation of the network using **mpdcm** and SPM. In this case, the range of the BOLD signals was $[-5.0, 3.4]$. The absolute difference between the signal simulated using Euler and Runge–Kutta integration schemes was always lower than 0.11 (max. mean absolute difference 0.013). Euler's method diverged always less than 0.11 from SPM (max. mean absolute difference 0.013), whereas the Runge–Kutta's method diverged less than 4×10^{-4} from SPM (mean absolute difference 9×10^{-6}). To compare all five sets of random parametrizations, we computed the variance of the difference between the predicted signal using Euler's and SPM's method, and Runge–Kutta's and SPM's method as a percentage of the total variance of the predicted signals. For simplicity, the total variance of the signal was computed using the simulations from SPM. We found that the variance of the difference from the predicted signals was never larger than 0.1% (Euler's method) and $10^{-5} \times 0.1\%$ (Runge–Kutta's method) of the total variance of the signal indicating that all three integration schemes resulted in highly similar simulations, with Runge–Kutta and SPM yielding virtually identical simulations. Finally, we considered the effect of floating point number representation on the simulations. In order to verify that single precision representation did not cause any systematic rounding errors, we repeated the simulations both under single and double precision and compared the relative error of the single precision implementation with respect to the double precision. The relative error was defined as

$$\frac{abs\,(double\,(svalue) - dvalue)}{abs\,(dvalue)} \tag{13}$$

where *svalue* and *dvalue* stand for the values obtained using single and double precision, respectively. We found that the maximal relative error did not exceed 2×10^{-5} for both SPM and **mpdcm**.

3.2. Performance gains

One of the main reasons for introducing **mpdcm** was to increase performance for costly simulations needed for inversion of DCM models. Here, performance on a three and six node network was assessed by running 20 consecutive simulations. The median values were used as summary statistics. Only minimal variance in the execution times was observed. Fig. 4 summarizes the results.

mpdcm can reduce the computation time required for generating simulations from DCM with respect to SPM by up to two orders of magnitude. It is important to note that although increasing the sampling rate of the input u has a linear effect on the performance of **mpdcm**, it has no effect on SPM's method. Thus, even though in practice the sampling rate of the input u was 16 times larger in the simulations generated with **mpdcm**, our toolbox performed better while preserving the nonlinearities of f. The difference between

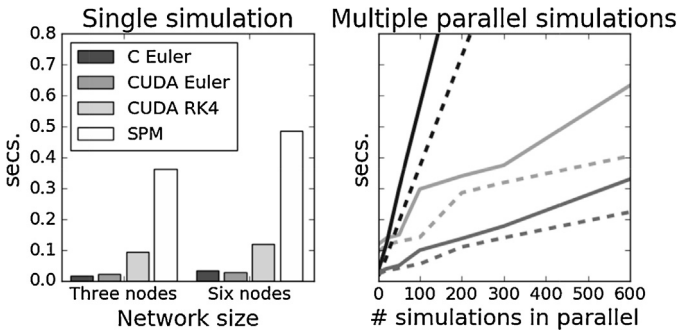

Fig. 4. Performance measurements. Left panel: Single simulation. Right panel: Multiple parallel simulations. The broken and solid lines correspond to DCMs with 3 and 6 nodes, respectively. Colors are matched to the left panel. Vertical axis is shared between panels. We found that mpdcm was faster integrating a single model (left panel) than the standard SPM integrator. This advantage scales with the number of regions, as the computation of f is performed in parallel. In the right panel, the time required for performing simulations in parallel is displayed as a function of the number of simultaneous parallel simulations. **mpdcm**'s Euler scheme was approximately 10 times faster (9.8 times for a 3 nodes network, and 9.9 times for a 6 node network) than the C implementation when 600 systems where integrated in parallel.

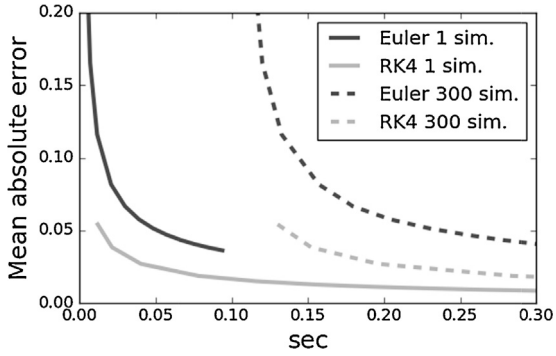

Fig. 6. Mean absolute error as a function of computation time in seconds. As expected, the error of the Euler method decreased slowly compared to Runge–Kutta's method.

the GPU implementation and the multithreaded CPU implementation is lower, with the GPU Euler's method being an order of magnitude faster than a CPU implementation when 600 simulations are performed in parallel. **mpdcm** is only marginally faster than a CPU implementation when a single system is evaluated.

We proceeded to investigate whether the performance of **mpdcm** was affected by the floating point number representation used. In Fig. 5, we compared the performance of **mpdcm** when using single and double precision floating point arithmetic in two different GPUs. In the *Nvidia* Geforce GTX 760, in the case of the Euler integrator, double precision was only marginally slower than single precision. However, double precision representation had a larger effect on Runge–Kutta's method. In particular, we found that for 600 parallel simulations, a double precision implementation was almost between two and three times slower than a single float implementation. This effect disappeared on the *Nvidia* 2050 M, where only minor differences between single and double precision arithmetic were found.

Finally, we considered the increases in accuracy obtained as a function of computation time. We compared Euler's and Runge–Kutta's method by simulating a six node network with the same parameters as those used in the previous section. However, input u was sampled at only 0.5 Hz while the time step h was varied from 1.0 s to 0.03514 s. A single simulation and 300 parallel and simultaneous simulations were compared. The mean absolute error was obtained against a simulation using Runge–Kutta's method with a time step of $h = 2^{-8}$. Results are displayed in Fig. 6.

Please note that these results should be considered only a rough indication of performance gains. Differences in performance depend on several parameters, such as the number of nodes, TR, sampling rates, input and output size and structure, and hardware. Also, in the case of CPU implementation the linear increase in time can of course be compensated by the use of a larger number of CPUs.

4. Applications

4.1. Thermodynamic integration

mpdcm was developed to increase the performance of DCM in the context of computationally demanding inference problems. One particularly interesting challenge is the evaluation of the model evidence or marginal likelihood of a DCM.

Comparison of alternative DCMs is usually done by comparing the marginal likelihood between models given a data set y. This corresponds to computing the Bayes factors between models:

$$\frac{p(y|m_1)}{p(y|m_2)} = \frac{\int p\left(y|\theta, m_1\right) p\left(\theta|m_1\right) \mathrm{d}\theta}{\int p\left(y|\theta, m_2\right) p\left(\theta|m_2\right) \mathrm{d}\theta}. \tag{14}$$

Although a powerful approach, exact computation of marginal likelihoods is usually not possible because almost all but the simplest models lead to intractable integrals in Eq. (14). For this reason, approximations such as the Akaike Information Criterion (AIC), the Bayesian Information Criterion (BIC) or the negative free energy are typically used (Penny et al., 2004). In addition, several sampling-based methods have been proposed in the literature to compute

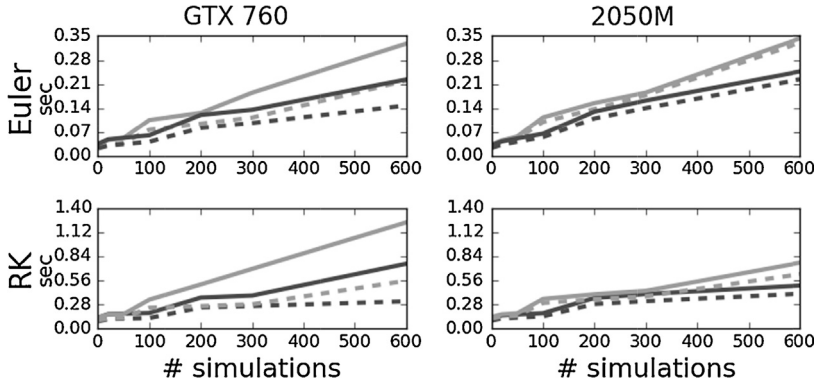

Fig. 5. Computation time in seconds as a function of the number of parallel simulations in two different Nvidia cards. Black and gray lines display DCMs of 3 and 6 regions. Solid and broken lines display results with double and single precision arithmetic, respectively. The effect of floating point representation was more pronounced in the GTX 760.

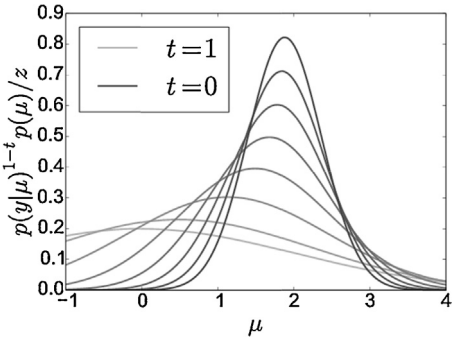

Fig. 7. Normalized power posterior densities $p\left(y\left|\theta,m\right.\right)^{1-t}p\left(\theta\left|m\right.\right)/z$. The likelihood and prior distributions are Gaussian. The prior distribution of μ is $N(0,\sqrt{2})$. y is a single observation with $y=2$. Eight different values of t are displayed. As the temperature decreases, the posterior distributions' variance decreases.

the evidence of intractable models (Kass and Raftery, 1995). For example, the identity

$$\int \frac{1}{p\left(y\left|\theta,m\right.\right)}p\left(\theta\left|y,m\right.\right)\mathrm{d}\theta=\frac{1}{p\left(y\left|m\right.\right)}, \tag{15}$$

suggests using samples from the posterior distribution to estimate the marginal likelihood, an approach called the posterior harmonic mean estimator. Although an asymptotically unbiased estimator, this approach has several shortcomings, including a potentially infinite variance (Raftery et al., 2007). Gelman and Meng (1998) introduced in the statistical literature an alternative method to estimate marginal likelihoods called thermodynamic integration, originally proposed by Kirkwood (1935). In this approach, the difference in the normalization constants between two distributions is computed by constructing a continuous path between them. By integrating along this path, it is possible to compute differences in marginal likelihoods. More formally, thermodynamic integration is based on the equivalence (or a more general formulation thereof):

$$\ln p\left(y\left|m\right.\right)=\ln\int p\left(y\left|\theta,m\right.\right)p\left(\theta\left|m\right.\right)\mathrm{d}\theta-\ln\int p\left(\theta\left|m\right.\right)\mathrm{d}\theta \tag{16}$$

$$=\int_1^0 E_{p(y|\theta)^{1-t}p(\theta)}\ln p\left(y\left|\theta,m\right.\right)\,\mathrm{d}t \tag{17}$$

The path between prior and posterior is constructed by weighting the log likelihood function by the temperature $1-t$ with $t\in[0,1]$. A graphical illustration of this path is displayed in Fig. 7. In practice this is implemented by first selecting a finite set of temperatures $0=t_1<t_2<\cdots<t_{N-1},<t_N=1$, then drawing samples from the unnormalized power posteriors $p\left(y\left|\theta,m\right.\right)^{1-t}p\left(\theta\left|m\right.\right)$, and finally numerically computing the integral in Eq. (17) using quadratures. Although it has been shown that in practice this method has higher accuracy and precision than alternative sampling methods (Calderhead and Girolami, 2009), it requires evaluating N different power posteriors, with its accuracy increasing with N.

Samples from the power posteriors can be obtained using the Metropolis–Hastings algorithm. In this case, the likelihood of a sample drawn from the proposal distribution can be used for the acceptance step. Since the computation of the likelihood is the most expensive step, **mpdcm** greatly diminishes the run time of this algorithm.

4.2. Parallel tempering

If thermodynamic integration is implemented using the Metropolis–Hastings algorithm, it can be extended to improve the mixing of the Markov chains by allowing parallel chains to exchange information without changing their stationary distribution. This method is referred to as "population MCMC", "replica exchange MCMC", or "parallel tempering" (see Sengupta et al., 2015). The basic intuition underlying this approach is the idea of a set of Markov chains $n=1,\ldots,N$ exchanging information in a way reminiscent of biological evolution, where chains in the population "mutate" and "mate" (Laskey and Myers, 2003). This intuition is formalized by the product distribution

$$\pi_1\left(\theta_1\right)\pi_2\left(\theta_2\right),\ldots,\pi_{N-1}\left(\theta_{N-1}\right)\pi_N\left(\theta_N\right), \tag{18}$$

where $\theta_1,\ldots,\theta_N$ represent a population of parameters. Swendsen and Wang (1986) originally proposed to condition the distributions $\pi_1,\ldots,\pi_N$ with a temperature parameter in the same way as in thermodynamic integration, i.e.,

$$\pi_n\left(\theta\right)\propto p\left(y|\theta,m\right)^{1-t_n}p\left(\theta|m\right), \tag{19}$$

with $0=t_1<\cdots<t_N=1$. In order to propagate information between chains, an *exchange* operator swaps the parameters θ_i with θ_j with probability:

$$\min\left(1,\frac{p\left(y\left|\theta_j,m\right.\right)^{1-t_i}p\left(y\left|\theta_i,m\right.\right)^{1-t_j}}{p\left(y\left|\theta_j,m\right.\right)^{1-t_j}p\left(y\left|\theta_i,m\right.\right)^{1-t_i}}\right). \tag{20}$$

Since the exchange is reversible, the stationary distribution of each chain does not change. In practice, samples from each chain are drawn from the distribution defined in Eq. (18) and then samples from the population are exchanged according to Eq. (20). Samples drawn using this method can be used for either parameter estimation or for computing the evidence of a model through thermodynamic integration. Informally, since chains at higher temperature can explore the parameter space more freely, parallel tempering can dramatically improve the mixing of each particular chain (Calderhead and Girolami, 2009).

Pseudocode for parallel tempering using **mpdcm** is provided in Algorithm 2 (Altekar et al., 2004). The implementation generalizes path sampling with independent chains by adding a swapping step. Using path sampling and parallel tempering has already been suggested in the context of DCM (Sengupta et al., 2015). Our goal here is to show that these algorithms can be easily and efficiently implemented through **mpdcm**.

Algorithm 2:
Requires: $M\geq 1$ (Number of samples)
Requires: $S\geq 1$ (Number of swaps)
Requires: $0=t_1\leq\cdots\leq t_N=1$ (Temperature schedule)
1: **for** $m=0$ **to** $M-1$ **do**
2: **for** $n=0$ **to** $N-1$ **do**
3: Sample $\theta_n^{(m)}$ *from* $p\left(y|u,\theta_n\right)^{1-t_n}p(\theta_n)$
4: **end for**
5: **for** $n=0$ **to** $N-1$ **in** parallel **do**
6: $Compute l_n^{(m)}=p\left(y\left|u,\theta_n^{(m)}\right.\right)$
7: **end for**
8: **for** $s=0$ **to** $S-1$ **do**
9: Randomly select k *with* $1\leq k\leq N-1$
with probability $\min\left(1,\frac{p\left(y|u,\theta_k^{(m)}\right)^{1-t_{k+1}}p\left(y\left|u,\theta_{k+1}^{(m)}\right.\right)^{1-t_k}}{p\left(y\left|u,\theta_k^{(m)}\right.\right)^{1-t_k}p\left(y\left|u,\theta_{k+1}^{(m)}\right.\right)^{1-t_{k+1}}}\right)$, *swap samples* $\theta_k^{(m)}$ *and* $\theta_{k+1}^{(m)}$
10: **end for**
11: **end for**
12: **for** $n=0$ **to** $N-1$ **do**
13: Compute $c_n=\frac{1}{M}\sum_{n=1}^{M}\ln l_n^{(m)}$
14: **end for**
15: **return** $\frac{1}{2}\sum_{n=1}^{N-1}(t_n-t_{n-1})(c_n+c_{n-1})$

Although the computational costs of parallel tempering are similar to those of path sampling with independent chains, information

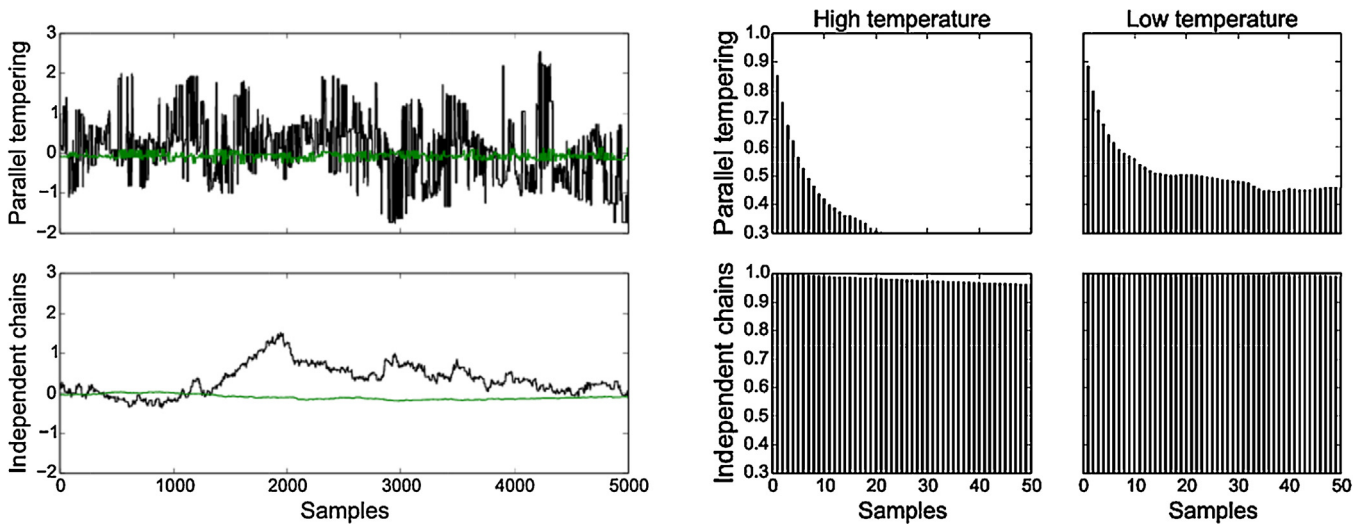

Fig. 8. Left: Samples from the same posterior distribution using independent chains and parallel tempering. Chains at low and high temperature are displayed in green and black respectively. Right: Autocorrelation function of the samples. In MCMC, samples are by definition correlated, although their stationary distribution corresponds to the target distribution. If samples are strongly correlated, more samples are necessary to estimate the moments of the target distribution. Since parallel tempering reduces the correlation between samples, the total number of necessary simulations diminishes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

exchange between chains makes the implementation of efficient parallel algorithms in computer grids far more challenging (Li et al., 2007). In particular, while path sampling can be implemented with completely parallel CPUs, parallel tempering requires the synchronization of the chains. Since **mpdcm** uses a single CPU and only the simulations are performed on a GPU, parallel tempering can be easily implemented. Fig. 8 displays samples of one connectivity parameter drawn using parallel tempering. The exchange operator effectively decorrelates samples drawn from each chain, increasing the statistical efficiency of this method at minimal computational costs.

Running parallel tempering using a purely CPU implementation is possible but the computational costs are much larger than with **mpdcm**. Assuming a DCM of 6 regions with 512 volumes, a total of 100 parallel chains, 40,000 iterations of the MCMC algorithm, and counting only the time required to perform simulations—based on results displayed in Fig. 4, it can be inferred that SPM would require over 22 days to finish the inversion of one session. For a multithreaded CPU implementation in C, the time required would be reduced to 6.3 h, and for **mpdcm** Euler's method, the time required would be reduced to 1.1 h.

Both thermodynamic integration as well as parallel tempering are implemented in **mpdcm** in the function tapas_*mpdcm_fmri_estimate.m*. Besides these two algorithms, we have implemented a maximum *a posteriori* estimator for single and multiple subjects and several other routines specified in the documentation.

5. Discussion

In this paper, we presented **mpdcm**, a toolbox aimed at speeding up simulations from the generative model that underlies DCM. Our implementation does not compromise the nonlinearities in the model and is nevertheless computationally efficient. We used two integration methods for generating simulations: Euler's method and a modified Runge–Kutta's fourth order method. The two integration schemes did not strongly differ from each other with regard to the simulated BOLD signal. They also did not strongly differ from the SPM's method based on Ozaki's scheme (Ozaki, 1992). However, they strongly differed in their performance (Fig. 4). Euler's method is more efficient because of the lower number of floating point operations and reduced memory complexity. Although more stable and precise methods exist, the characteristics of typical fMRI experiments and of DCM make Euler's method an acceptable alternative when precision is strongly outweighed by the need of a large number of simulations. Highly precise simulations can still be generated using Runge–Kutta's method without large additional computational time needed. Performance differences between a multithreaded CPU implementation of Euler's method and **mpdcm** reached one order of magnitude, similarly to benchmarks published before for other algorithms (Lee et al., 2010b).

Higher order integrators were not considered here, as integrators of order 5 or higher increase the memory needs of most algorithms. Low storage explicit Runge–Kutta methods are available (Williamson, 1980; Ketchenson, 2010), and we implemented Blum's (1962) modified Runge–Kutta's fourth order algorithm, which requires only 3 registers for each state variable. Although not presented here, our toolbox includes an adaptive size integrator based on Bogacki and Shampine (1989) third order, embedded algorithm. This scheme has several computational advantages, and adaptive size integrators tend to outperform fixed step implementations. However, by nature, the number of iterations of this algorithm is unknown at scheduling time, a characteristic that makes the allocation of GPU resources much less efficient.

The double precision floating point format as defined by the IEEE 754 standard (see Goldberg, 1991) is the default data type of several scientific applications. However, it is not always clear whether the advantages inherent to this format are necessary in all applications. Double precision floats provide a dynamic range of over 600 orders of magnitude, a range which is not needed in, for example, brain imaging. Moreover, not all applications require more precision than a small number of decimal places, while double precision format provides approximately 15 decimal places of precision. Regarding performance, in the case of $\times$86 architectures, double and single precision arithmetic is implemented by the same 80 bit floating point unit. Therefore, in $\times$86 architectures, the main performance difference between both formats is due to memory latencies and bandwidth. In contrast to this, in the case of consumer level GPUs the theoretical ratio between peak double and single precision FLOPS is often close to 1:24 or 1:32. We found that double precision arithmetic (compared to single precision) reduced

performance approximately by a factor of 3 in a GPU with a very low double to single floating point peak performance ratio (1:32). This difference was reduced to roughly 1:2 in a GPU designed for professional purposes.

The performance comparison between double and single precision indicates that **mpdcm** is largely memory bounded, as the difference in performance between the double and single precision performance are far from the differences between peak FLOPS. Importantly, we did not find any notable effects of rounding-off errors in the single precision implementation of **mpdcm,** with the highest difference being far below the signal to noise ratio that is typical of fMRI experiments. Single precision floating point arithmetic is thus sufficiently accurate and recommended for **mpdcm,** as the effect on the performance of Runge Kutta's method is significant. This is particularly important for consumer-level GPUs, where the peak single precision arithmetic performance is much larger than the double precision peak performance.

mpdcm is particularly attractive when several systems are simulated in parallel, because under those conditions performance gains of up to two orders of magnitude with respect to SPM and one order of magnitude with respect to a multithreaded *C* implementation become evident. In order to show how to exploit these large increases in performance, we presented an algorithm that can be easily optimized with our toolbox. In particular, we presented a GPU implementation of parallel tempering, a variant of MCMC that is known to improve the mixing properties of the Markov chains of the Metropolis–Hastings algorithm, reducing the computational costs with respect to thermodynamic integration with independent chains. Implementing parallel tempering using **mpdcm** is straightforward, as all the parallelization is managed internally at the GPU level. Although the computational gains obtained through our toolbox have been illustrated using sampling algorithms, variational methods that require numerical optimization can also profit, as numerical differentiation can be easily implemented in **mpdcm** (but see Sengupta et al., 2014). Moreover, while we stressed here that **mpdcm** can generate simulations from models with the same input and different parametrizations, it also supports generating simulations from different models *and* different inputs. This is important, as it facilitates inference in models that incorporate several data sets, such as hierarchical multi-subject models (Raman et al. unpublished results).

We have presented two sampling algorithms that exploit the massive parallelism allowed by **mpdcm.** Other sampling methods that make use of parallel simulations have been proposed. For example, multiple try MCMC (Liu et al., 2000) relies on proposing several samples instead of a single one, as usually done in the Metropolis–Hastings algorithm. More recently, Calderhead (2014) proposed an algorithm similar in spirit that extends the acceptance step by using a finite state Markov chain. The algorithms presented here have the advantage of allowing the estimation of the model evidence, and, in the case of parallel tempering, improving the mixing of all the chains. It is important to note that this type of algorithms can be potentially combined, in order to fully exploit the gains in parallel, simultaneous evaluations of the model. Thus, parallel simulations can be added to increase the statistical efficiency of an algorithm at a minimal cost.

Several authors have proposed before to use GPUs in the context of MCMC methods (for a review see Guo, 2012; Suchard et al., 2010). For example, Lee et al. (2010a) considered the improvement in mixing rate in Gaussian mixture models using parallel tempering and found a large increase in statistical efficiency. Jacob et al. (2011) proposed to use parallelism to increase the efficiency of an independent Metropolis–Hastings algorithm. In this variant of MCMC, the updates proposed do not depend on the former step of the chain. Thus, candidate updates can be precomputed in parallel. More generally, the increased computational power provided by GPU has opened the door to Bayesian methods based on sampling that were unfeasible before.

In summary, we envisage that future applications of DCM will require increasingly large computational resources, for example, when the likelihood landscape of a model is multimodal, or when implementing sampling methods for multi-subject analysis. Due to the wide availability of GPUs, algorithmic advances can be readily available in the absence of access to large computer grids, or when there is a lack of expertise to implement algorithms in such grids.

Software note

The **mpdcm** toolbox is available under the GPL license as part of the open source TAPAS software at www.translationalneuromodeling.org/software.

Acknowledgments

K.E.S. is supported by the René and Susanne Braginsky Foundation. K.E.S. and S.R. are supported by the Clinical Research Priority Program "Multiple Sclerosis" and "Molecular Imaging" at the University of Zurich. W.P. is supported by a core grant from the Wellcome Trust (091 593/Z/10/Z). B.S. is supported by the Wellcome Trust (088130/Z/09/Z).

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.jneumeth.2015.09.009.

References

Altekar G, Dwarkadas S, Huelsenbeck JP, Ronquist F. Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. Bioinformatics 2004;20(Feb (3)):407–15.
Blum EK. A modification of the Runge–Kutta fourth-order method. Math Comput 1962;16(78):176–87.
Butcher JC. Numerical methods for ordinary differential equations. 2nd ed. Chichester: John Wiley & Sons Ltd; 2008.
Buxton RB, Wong EC, Frank LR. Dynamics of blood flow and oxygenation changes during brain activation: the balloon model. Magn Reson Med 1998;39(6):855–64.
Bogacki P, Shampine LF. A 3 (2) pair of Runge–Kutta formulas. Appl Math Lett 1989;2(4):321–5.
Calderhead B. A general construction for parallelizing Metropolis–Hastings algorithms. Proc Natl Acad Sci USA 2014;111(Dec (49)):17408–13.
Calderhead B, Girolami M. Estimating Bayes factors via thermodynamic integration and population MCMC. Comput Stat Data Anal 2009;53(12):4028–45.
Chumbley JR, Friston KJ, Fearn T, Kiebel SJ. A Metropolis–Hastings algorithm for dynamic causal models. Neuroimage 2007;38(Nov (3)):478–87.
Eklund A, Dufort P, Forsberg D, LaConte SM. Medical image processing on the GPU—past, present and future. Med Image Anal 2013;17(Dec (8)):1073–94.
Eklund A, Dufort P, Villani M, Laconte S. BROCCOLI: software for fast fMRI analysis on many-core CPUs and GPUs. Front Neuroinform 2014;8:8–24.
Friston K, Mattout J, Trujillo-Barreto N, Ashburner J, Penny W. Variational free energy and the Laplace approximation. Neuroimage 2007;34(1):220–34.
Friston KJ. Bayesian estimation of dynamical systems: an application to fMRI. Neuroimage 2002;16(Jun (2)):513–30.
Friston KJ, Harrison L, Penny W. Dynamic causal modelling. Neuroimage 2003;19(Aug (4)):1273–302.
Ferreira da Silva AR. Cudabayesreg: parallel implementation of a Bayesian multilevel model for fmri data analysis. J Stat Softw 2011;44(4):1–24 (10).
Gelman A, Meng X. L. Simulating normalizing constants: from importance sampling to bridge sampling to path sampling. Stat Sci 1998;13(2):163–85.
Gelman A, Carlin JB, Stern HS, Rubin DB. Bayesian data analysis. London: Chapman and Hall/CRC; 2003.
Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Comput Surv (CSUR) 1991;23(1):5–48.
Guo G. Parallel statistical computing for statistical inference. J Stat Theory Pract 2012;6(3):536–65.
Hernandez M, Guerrero GD, Cecilia JM, Garcia JM, Inuggi A, Jbabdi S, et al. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs. PLoS ONE 2013;8(4):e61892.

Jacob P, Robert CP, Smith MH. Using parallel computation to improve independent Metropolis–Hastings based estimation. J Comput Graph Stat 2011;20(3):616–35.
Jing Y, Zeng W, Wang N, Ren T, Shi Y, Yin J, et al. GPU-based parallel group ICA for functional magnetic resonance data. Comput Methods Programs Biomed 2015;119(Apr (1)):9–16.
Kass RE, Raftery AE. Bayes factors. J Am Stat Assoc 1995;90(430):773–95.
Ketchenson DI. Runge–Kutta's methods with minimum storage implementations. J Comput Phys 2010;229(5):1763–73.
Kirkwood JG. Statistical mechanics of fluid mixtures. J Chem Phys 1935;3(5):300–13.
Laskey KB, Myers JW. Population Markov chain Monte Carlo. Mach Learn 2003;50(1–2):175–96.
Lee A, Yau C, Giles MB, Doucet A, Holmes CC. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. J Comput Graph Stat 2010a;Dec (19)(4):769–89.
Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, et al. Debunking the 100× GPU vs. CPU Myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput Archit News 2010b;38(Jun (3)):451–60.
Li Y, Mascagni M, Gorin A. Decentralized replica exchange parallel tempering: an efficient implementation of parallel tempering using MPI and SPRNG. In: Proceedings of international conference on computational science and its applications (ICCSA). Kuala Lumpur: Springer; 2007. p. 507–19.
Liu JS, Liang F, Wong WH. The multiple-try method and local optimization in metropolis sampling. J Am Stat Assoc 2000;95(449):121–34.
MacKay DJC. Information theory, inference, and learning algorithms. Cambridge: Cambridge University Press; 2003.
Ozaki T. A bridge between nonlinear time series models and nonlinear stochastic dynamical systems: a local linearization approach. Stat Sin 1992;2(1):113–35.
Penny WD, Stephan KE, Mechelli A, Friston KJ. Comparing dynamic causal models. Neuroimage 2004;22(Jul (3)):1157–72.
Penny WD, Stephan KE, Daunizeau J, Rosa MJ, Friston KJ, Schoeld TM, et al. Comparing families of dynamic causal models. PLoS Comput Biol 2010;6(3):e1000709.
Raimondo F, Kamienkowski JE, Sigman M, Fernandez Slezak D. Cudaica: Gpu optimization of infomax-ica eeg analysis. Comput Intell Neurosci 2012;2012:2.
Raftery AE, Newton MA, Satagopan JM, Krivitsky PN. Estimating the integrated likelihood via posterior simulation using the harmonic mean identity. In: Bernardo JM, Bayarri MJ, Berger JO, Dawid AP, Heckerman D, Smith AFM, West M, editors. Bayesian Statistics, 8. Oxford: Oxford University Press; 2007. p. 1–45.
Sengupta B, Friston KJ, Penny WD. Efficient gradient computation for dynamical models. Neuroimage 2014;98(Sep):521–7.
Sengupta B, Friston KJ, Penny WD. Gradient-free MCMC methods for dynamic causal modelling. Neuroimage 2015;112(May):375–81.
Shi L, Liu W, Zhang H, Xie Y, Wang D. A survey of GPU-based medical image computing techniques. Quant Imaging Med Surg 2012;2(Sep (3)):188–206.
S. Raman, L. Deserno, F. Schlagenhauf, K.E. Stephan. A hierarchical model for unifying unsupervised generative embedding and empirical Bayes, unpublished results.
Stephan KE, Weiskopf N, Drysdale PM, Robinson PA, Friston KJ. Comparing hemodynamic models with DCM. Neuroimage 2007;38(Nov (3)):387–401.
Stephan KE, Kasper L, Harrison LM, Daunizeau J, den Ouden HEM, Breakspear M, et al. Nonlinear dynamic causal models for fMRI. Neuroimage 2008;42(Aug):649–62.
Suchard MA, Wang Q, Chan C, Frelinger J, Cron A, West M. *Understanding* GPU programming for statistical computation: studies in massively parallel massive mixtures. J Comput Graph Stat 2010;19(Jun (2)):419–38.
Swendsen R, Wang J. Replica Monte Carlo simulation of spin-glasses. Phys Rev Lett 1986;57(Nov):2607–9.
Wang WJ, Hsieh IF, Chen CC. Accelerating computation of DCM for ERP in MATLAB by external function calls to the GPU. PLoS ONE 2013;8(6), e66599.
Williamson JH. Low-storage Runge–Kutta schemes. J Comput Phys 1980;35(1):48–56.