WORKING PAPER

Parallel sequential Monte Carlo for efficient density combination: The DeCo MATLAB toolbox

NORGES BANK RESEARCH

11 | 2014

AUTHORS:
ROBERTO CASARIN

STEFANO GRASSI

FRANCESCO RAVAZZOLO

HERMAN K. VAN DIJK

NORGES BANK

Working papers fra Norges Bank, fra 1992/1 til 2009/2 kan bestilles over e-post:

servicesenter@norges-bank.no

Fra 1999 og senere er publikasjonene tilgjengelige på www.norges-bank.no

Working papers inneholder forskningsarbeider og utredninger som vanligvis ikke har fått sin endelige form. Hensikten er blant annet at forfatteren kan motta kommentarer fra kolleger og andre interesserte. Synspunkter og konklusjoner i arbeidene står for forfatternes regning.

Working papers from Norges Bank, from 1992/1 to 2009/2 can be ordered by e-mail

servicesenter@norges-bank.no

Working papers from 1999 onwards are available on www.norges-bank.no

Norges Bank's working papers present research projects and reports (not usually in their final form) and are intended inter alia to enable the author to benefit from the comments of colleagues and other interested parties. Views and conclusions expressed in working papers are the responsibility of the authors alone.

ISSN 1502-8143 (online)
ISBN 978-82-7553-818-3 (online)

Parallel Sequential Monte Carlo for Efficient Density Combination: The DeCo MATLAB Toolbox*

Roberto Casarin
University Ca' Foscari of Venice
and GRETA

Stefano Grassi
School of Economics
University of Kent and CREATES

Francesco Ravazzolo
Norges Bank
and BI Norwegian Business School

Herman K. van Dijk
Erasmus University Rotterdam
VU University Amsterdam
and Tinbergen Institute

Abstract

This paper presents the MATLAB package **DeCo** (density combination) which is based on the paper by Billio, Casarin, Ravazzolo, and van Dijk (2013) where a constructive Bayesian approach is presented for combining predictive densities originating from different models or other sources of information. The combination weights are time-varying and may depend on past predictive forecasting performances and other learning mechanisms. The core algorithm is the function **DeCo** which applies banks of parallel Sequential Monte Carlo algorithms to filter the time-varying combination weights. The **DeCo** procedure has been implemented both for standard CPU computing and for graphical process unit (GPU) parallel computing. For the GPU implementation we use the MATLAB parallel computing toolbox and show how to use general purpose GPU computing almost effortlessly. This GPU implementation provides a speed up of the execution time of up to seventy times on a standard CPU MATLAB implementation on a multicore CPU. We show the use of the package and the computational gain of the GPU version through some simulation experiments and empirical applications.

Keywords: Density forecast combination, sequential Monte Carlo, parallel computing, GPU, MATLAB.

1. Introduction

Combining forecasts from different statistical models or other sources of information is a crucial issue in many different fields of science. Several papers have been proposed to handle this issues with Bates and Granger (1969) as one of the first attempts in this field. Initially the focus was on defining

*This working paper should not be reported as representing the views of Norges Bank. The views expressed are those of the authors and do not necessarily reflect those of Norges Bank. For their useful comments, we thank Daniel Armyr and Sylvia Frühwirth-Schnatter and seminar and conference participants at the European Seminar on Bayesian Econometrics 2013, Norges Bank, the 7th Rimini Bayesian Econometrics Workshop and the 2013 Vienna IHS Time-Series Workshop. Roberto Casarin's research is supported by the Italian Ministry of Education, University and Research (MIUR) PRIN 2010-11 grant, and by funding from the European Union, Seventh Framework Programme FP7/2007-2013 under grant agreement SYRTO-SSH-2012-320270.

and estimating combination weights for point forecasting. For instance, Granger and Ramanathan (1984) propose to combine forecasts with unrestricted least squares regression coefficients as weights. Terui and van Dijk (2002) generalize least squares weights by specifying the weights in the dynamic forecast combination as a state space model with time-varying weights that are assumed to follow a random walk process. Recently, research interest has shifted to the construction of combinations of predictive *densities* (and not point forecasts) as well as allowing for model set *incompleteness* (the true model may not be included in the set of models for prediction) and *learning*. Further, different *model evaluation* criteria are used. Hall and Mitchell (2007) and Geweke and Amisano (2010) propose using combination schemes based on the Kullback-Leibler score; Gneiting and Raftery (2007) recommend strictly proper scoring rules, such as the Cumulative Rank Probability Score, in particular, if the focus is on some particular area, such as extreme tails, of the distribution. Billio *et al.* (2013) (hereby BCRVD (2013)) provide a general Bayesian distributional state space representation of predictive densities and specify combination schemes that allow for an incomplete set of models and different learning mechanisms and scoring rules.

The design of algorithms for a numerically efficient combination remains a challenging issue (e.g., see Gneiting and Raftery 2007). BCRVD (2013) propose a combination algorithm based on Sequential Monte Carlo filtering. The proposed algorithm makes use of a random grid from the set of predictive densities and runs a particle filter at each point of the grid. The procedure is computationally intensive when the number of models to combine increases. A contribution of this paper is to present a MATLAB (see The MathWorks, Inc. (2011)) package **DeCo** (Density Combination) for the combination of predictive densities and a simple Graphical User Interface (GUI) for the use of this package.

This paper provides, through the **DeCo** package, an efficient implementation of BCRVD (2013) algorithm based on CPU and GPU parallel computing. We make use of recent increases in computing power and recent advances in parallel programming techniques. The focus of the microprocessor industry, mainly driven by Intel and AMD, has shifted from maximizing the performance of a single core to integrating multiple cores in one chip, see Sutter (2005) and Sutter (2011). Contemporaneously, the needs of the video game industry, requiring increasing computational performance, boosted the development of the Graphics Processing Unit (GPU), which enabled massively parallel computation.

In the present paper, we follow the recent trend of using GPUs for general, non-graphics, applications (prominently featuring those in scientific computing), the so-called general-purpose computing on graphics processing unit (GPGPU). The GPGPU has been applied successfully in different fields such as astrophysics, biology, engineering, and finance, where quantitative analysts started using this technology well ahead of academic economists, see Morozov and Mathur (2011) for a literature review.

To date, the adoption of GPU computing technology in economics and econometrics has been relatively slow compared with other fields. There are a few papers that deal with this interesting topic, see Suchard, Holmes, and West (2010), Morozov and Mathur (2011), Aldrich, Fernández-Villaverde, Gallant, and Rubio Ramırez (2011), Geweke and Durham (2012), Dziubinski and Grassi (2013) and Creel, Mandal, and Zubair (2012). This is odd given the fact that parallel computing in economics has a long history. An early attempt to use parallel computation for Monte Carlo simulation is Chong and Hendry (1986), while Swann (2002) develops parallel implementation of maximum likelihood estimation. Creel and Goffe (2008) discuss a number of economic and econometric problems where parallel computing can be applied. The low diffusion of this technology in economics and econometrics, according to Creel (2005), is mainly due to two issues, which are the high cost of the hardware, e.g. cluster, and the steep learning curve of dedicated programming languages as CUDA (Compute Unified Device Architecture, see Nvidia Corporation (2010)), OpenCL (Khronos OpenCL Working Group (2009)), Thrust (Hoberock and Bell (2011)) and C++ AMP (C++ Accelerated Massive Par-

	Advantages	Disadvantages
CUDA	Free	Vendor Lock-in
OpenCL	Free	Difficult to program
	Heterogeneous	
Thrust	Free	Vendor Lock-in
	Easy to program	
C++ AMP	Open Standard	Currently only Windows implementations exist
	Heterogeneous	
	Free (Express Edition)	
	Easy to program	

Table 1: Comparison of different currently available GPGPU approaches.

allelism, see Gregory and Miller (2012)). Table 1 compares different currently available GPGPU approaches. The recent increase in attention to parallel computing is motivated by the fact that the hardware costs issue has been solved by the introduction of modern GPUs with relatively low cost. Nevertheless, the second issue remains open. For example, Lee, Christopher, Giles, Doucet, and Holmes (2010) report that a programmer proficient in C (Press, Teukolsky, Vetterling, and Flannery (1992)) or C++ (Stroustrup (2000)), a programming skill that can take some time to be acquired, should be able to code effectively in CUDA within a few weeks.

We aim to contribute to this stream of literature by showing that GPU computing can be carried out almost without any extra effort using the parallel toolbox of MATLAB (available in version 2012b and following releases, see The MathWorks, Inc. (2011)) and a suitable approach to MATLAB coding of the algorithms. The MATLAB environment allows easy use of GPU programming without learning CUDA. We emphasize that this paper is not intended to compare CPU and GPU computing. In fact, we propose the combination algorithm for both standard parallel CPU and for parallel GPU computation. Our simulation and empirical experiments show that the **DeCo** GPU version is faster 3 to 10 times than the parallel multi-core CPU version, similar to recommendations in Brodtkorb, Hagen, and Saetra (2013), and up to 70 times faster than the standard sequential CPU version.

The structure of the paper is as follows. Section 2 introduces the principles of density forecast combinations with time-varying weights and parallel sequential Monte Carlo algorithms. Section 3 presents a parallel sequential Monte Carlo algorithm for density combinations. It also provides background material on GPU computing in MATLAB. Section 4 carry out a comparison, using a Monte Carlo, between CPU and GPU calculation. Section 5 reports the results for the macroeconomic empirical application. Section 6 concludes. Appendix A describes the structure of the algorithm and Appendix B shows the package GUI.

2. Time-varying combinations of predictive densities

2.1. A combination scheme

BCRVD (2013) introduces a general density combination scheme, which allows for time-varying weights; model set incompleteness (meaning the true model might not be in the model set); combination weight uncertainty and learning. The authors give a general distributional representation of the combination, provide an effective algorithm for the sequential estimation of the weights and discuss some alternative specifications of the combination and of weight dynamics. In the package, and the

paper, we apply for convenience the Gaussian combination scheme with logistic weights applied by BCRVD (2013).

Let us denote with $\mathbf{v}_{1:t} = (\mathbf{v}_1, \ldots, \mathbf{v}_t)$ a collection of vectors $\mathbf{v}_s$ with $s = 1, \ldots, t$. Let $\mathbf{y}_t \in \mathcal{Y} \subset \mathbb{R}^L$ be the L-vector of observable variables at time t and $\tilde{\mathbf{y}}_t = (\tilde{\mathbf{y}}'_{1,t}, \ldots, \tilde{\mathbf{y}}'_{K,t})' \in \mathcal{Y}^K$, with element $\tilde{\mathbf{y}}_{k,t} = (\tilde{y}^1_{k,t}, \ldots, \tilde{y}^L_{k,t})' \in \mathcal{Y} \subset \mathbb{R}^L$ the typical one-step-ahead predictor for $\mathbf{y}_t$ for the k-th model, $k = 1, \ldots, K$, in the pool, with predictive density $p(\tilde{\mathbf{y}}_{k,t}|\mathbf{y}_{1:t-1})$. The combination scheme is specified as:

$$p(\mathbf{y}_t|W_t, \tilde{\mathbf{y}}_t) \propto |\Sigma|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{y}_t - W_t\tilde{\mathbf{y}}_t)' \Sigma^{-1} (\mathbf{y}_t - W_t\tilde{\mathbf{y}}_t)\right\} \tag{1}$$

$t = 1, \ldots, T$, where $W_t = (\mathbf{w}^1_t, \ldots, \mathbf{w}^L_t)'$ is a weight matrix, with $\mathbf{w}^l_t = (w^l_{1,t}, \ldots, w^l_{KL,t})'$ as the l-th row vector containing the combination weights for the KL elements of $\tilde{\mathbf{y}}_t$ and for the prediction of $y_{l,t}$. In this paper we assume diagonal covariance matrix, i.e. $\Sigma = \text{diag}\{\sigma^2_1, \ldots, \sigma^2_L\}$.

The dynamics of the combination weights $w^l_{h,t}$ is

$$w^l_{h,t} = \frac{\exp\{x^l_{h,t}\}}{\sum_{j=1}^{KL} \exp\{x^l_{j,t}\}}, \quad \text{with } h = 1, \ldots, KL \tag{2}$$

where

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}) \propto |\Lambda|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x}_t - \mathbf{x}_{t-1})' \Lambda^{-1} (\mathbf{x}_t - \mathbf{x}_{t-1})\right\} \tag{3}$$

with $\mathbf{x}_t = \text{vec}(X_t) \in \mathcal{X} \subset \mathbb{R}^{KL^2}$ where $X_t = (\mathbf{x}^1_t, \ldots, \mathbf{x}^L_t)'$. In this paper we assume the covariance matrix of the state noise is diagonal, i.e. $\Lambda = \text{diag}\{\lambda_1, \ldots, \lambda_{KL^2}\}$. A learning mechanism can also be added to the weight dynamics, resulting in:

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{y}_{t-\tau:t-1}, \tilde{\mathbf{y}}_{t-\tau:t-1}) \propto |\Lambda|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x}_t - \boldsymbol{\mu}_t)' \Lambda^{-1} (\mathbf{x}_t - \boldsymbol{\mu}_t)\right\} \tag{4}$$

where $\boldsymbol{\mu}_t = \mathbf{x}_{t-1} - \Delta\mathbf{e}_t$, $\Delta\mathbf{e}_t = (\mathbf{e}_t - \mathbf{e}_{t-1})$ and elements of $\mathbf{e}_t$

$$e^l_{K(l-1)+k,t} = (1-\lambda)\sum_{i=1}^{\tau} \lambda^{i-1} f\left(y^l_{t-i}, \tilde{y}^l_{k,t-i}\right),$$

$k = 1, \ldots, K$, $l = 1, \ldots, L$, with $\lambda \in [0, 1]$ a discount factor and τ the number of previous observations used in the learning. We assume the function $f()$ defines a learning strategy. Note that the **DeCo** package relies on a general algorithm which can account for different scoring rules, such as the Kullback-Leibler score (Hall and Mitchell (2007) and Geweke and Amisano (2010)) and the Cumulative Rank Probability Score (Gneiting and Raftery (2007)).[1]

The proposed state space representation of the combination scheme composed by the observation equation in (1) and the latent equations (2), (3) or (4) provides a forecast density for the observable variables, conditional on the predictors and on the combination weights. Moreover, the representation is quite general, allowing for nonlinear and non-Gaussian combination models. We use sequential Monte Carlo algorithms, also known as particle filters, to estimate sequentially over time the optimal combination weights and the predictive density.

[1]The user interested in different scoring rules should change the command lines with the learning in the file PFCoreGPU.m for the GPU version and PFCore.m for the CPU version.

The steps of the density combination algorithm are sketched in the rest of this section. Let $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^{n_\theta}$ be the parameter vector of the combination model, with $\boldsymbol{\theta} = (\log \sigma_1^2, \ldots, \log \sigma_L^2, \log \lambda_1, \ldots, \log \lambda_{KL^2})$. Define the augmented state vector $\mathbf{z}_t = (\mathbf{x}_t, \boldsymbol{\theta}_t) \in \mathcal{Z}$ and the augmented state space $\mathcal{Z} = \mathcal{X} \times \Theta$, where $\boldsymbol{\theta}_t = \boldsymbol{\theta}$, $\forall t$. The distributional state space form of the forecast model is

$$\mathbf{y}_t \sim p(\mathbf{y}_t|\mathbf{z}_t, \tilde{\mathbf{y}}_t) \quad \text{(measurement density)} \tag{5}$$

$$\mathbf{z}_t \sim p(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{y}_{1:t-1}, \tilde{\mathbf{y}}_{1:t-1}) \quad \text{(transition density)} \tag{6}$$

$$\mathbf{z}_0 \sim p(\mathbf{z}_0) \quad \text{(initial density)} \tag{7}$$

The state predictive and filtering densities conditional on the predictive variables $\tilde{\mathbf{y}}_{1:t}$ are

$$p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) = \int_{\mathcal{Z}} p(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) p(\mathbf{z}_t|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) d\mathbf{z}_t \tag{8}$$

$$p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t+1}, \tilde{\mathbf{y}}_{1:t+1}) = \frac{p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}, \tilde{\mathbf{y}}_{t+1}) p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t})}{p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t})} \tag{9}$$

respectively, which represent the optimal nonlinear filter (see Doucet, Freitas, and Gordon (2001)). The marginal predictive density of the observable variables is then

$$p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}) = \int_{\mathcal{Y}} p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{t+1}) p(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t}) d\tilde{\mathbf{y}}_{t+1} \tag{10}$$

where

$$p(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t}) = \prod_{k=1}^{K} p(\tilde{\mathbf{y}}_{k,t+1}|\mathbf{y}_{1:t}) \tag{11}$$

is the joint density of the predictors and $p(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{t+1})$ is defined as

$$\int_{\mathcal{Z} \times \mathcal{Y}^{Kt}} p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}, \tilde{\mathbf{y}}_{t+1}) p(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}) p(\tilde{\mathbf{y}}_{1:t}|\mathbf{y}_{1:t-1}) d\mathbf{z}_{t+1} d\tilde{\mathbf{y}}_{1:t} \tag{12}$$

and represents the conditional predictive density of the observable given the past values of the observable and of the predictors.

2.2. A combination algorithm

The analytical solution of the optimal filter for non-linear state space models is generally not known. An approximate solution is needed. We apply a numerical approximation method, which converges to the optimal filter in Hilbert metric, in the total variation norm and in a weaker distance suitable for random probability distributions (e.g., see Legland and Oudjane (2004)). More specifically, we consider a sequential Monte Carlo (SMC) approach to filtering. See Doucet *et al.* (2001) for an introduction to SMC and Creal (2009) for a recent survey on SMC in economics. We propose using banks of SMC filters, where each filter is conditioned on a sequence of realizations of the predictor vector $\tilde{\mathbf{y}}_t$, see BCRVD(2013). The resulting algorithm for the sequential combination of densities is defined through the following steps, see Section A for a graphical representation.

Step 0 *(initialization, eq. (7))*

Initialize independent particle sets $\Xi_0^j = \{\mathbf{z}_0^{i,j}, \omega_0^{i,j}\}_{i=1}^N$, $j = 1, \ldots, M$. Each particle set Ξ_0^j contains N i.i.d. random variables $\mathbf{z}_0^{i,j}$ with random weights $\omega_0^{i,j}$.[2] Initialize a random grid over the set of

[2]The parameter $\boldsymbol{\theta}_0^{i,j}$ in $\mathbf{z}_0^{i,j}$ can be fixed or estimated. When estimated, the parameter must be initialized by drawing from the prior. The prior requires the specification of the mean and the standard deviation of the random noise of the log normal random walk process that $\boldsymbol{\theta}$ is assumed to follow. DeCo toolbox includes all these options, see Section B .

predictors, by generating i.i.d. samples $\tilde{\mathbf{y}}_1^j$, $j = 1, \ldots, M$, from $p(\tilde{\mathbf{y}}_1|\mathbf{y}_0)$. We use the sample of observations $\mathbf{y}_0$ to initialize the individual predictors.

Step 1 (predictor generation, eq. (11))

At the iteration $t+1$ of the combination algorithm, we approximate the predictive density $p(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t})$ with the discrete probability

$$p_M(\tilde{\mathbf{y}}_{t+1}|\mathbf{y}_{1:t}) = \sum_{j=1}^{M} \delta_{\tilde{\mathbf{y}}_{t+1}^j}(\tilde{\mathbf{y}}_{t+1})$$

where $\tilde{\mathbf{y}}_{t+1}^j$, $j = 1, \ldots, M$, are i.i.d. samples from the predictive densities and $\delta_x(y)$ denotes the Dirac mass centered at x. This approximation is also motivated by the forecasting practice (see Jore, Mitchell, and Vahey (2010)). The predictions usually come, from different models or sources, in the form of discrete densities. In some cases, this is the result of a collection of point forecasts from many subjects, such as surveys forecasts. In other cases the discrete predictive is a result of a Monte Carlo approximation of the predictive density (e.g., importance sampling or Markov chain Monte Carlo approximation of the model predictive density).

Step 2 (filtering and prediction)

We assume an independent sequence of particle sets $\Xi_t^j = \{\mathbf{z}_{1:t}^{i,j}, \omega_t^{i,j}\}_{i=1}^N$, $j = 1, \ldots, M$, is available at time $t+1$ and that each particle set provides the approximation

$$p_{N,j}(\mathbf{z}_t|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}^j) = \sum_{i=1}^{N} \omega_t^{i,j} \delta_{\mathbf{z}_t^{i,j}}(\mathbf{z}_t) \tag{13}$$

of the filtering density, $p(\mathbf{z}_t|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}^j)$, conditional on the j-th predictor realization, $\tilde{\mathbf{y}}_{1:t}^j$. Then M independent SMC algorithms are used to find a new sequence of M particle sets, which include the information available from the new observation and the new predictors. Each SMC algorithm iterates, for $j = 1, \ldots, M$, the following steps.

Step 2.a (state prediction, eq. (8))

The basic SMC algorithm uses the particle set to approximate the predictive density with an empirical density. We use a regularized version of the SMC procedure (e.g., see Liu and West (2001), Musso, Oudjane, and Legland (2001) and Casarin and Marin (2009)). More specifically, the predictive density of combination weights and parameters, $\mathbf{z}_{t+1}$, conditional on $\tilde{\mathbf{y}}_{1:t}^j$ and $\mathbf{y}_{1:t}$ is approximated as follows

$$p_{N,j}(\mathbf{z}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}^j) = \sum_{i=1}^{N} p(\mathbf{x}_{t+1}|\mathbf{x}_t, \boldsymbol{\theta}_{t+1}, \mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}^j) \omega_t^{i,j} \delta_{\mathbf{x}_t^{i,j}}(\mathbf{x}_t) K_h(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t^{i,j}) \tag{14}$$

where $K_h(y) = h^{-n_\theta} K(y/h)$ is the regularization kernel, K being a positive function defined on $\mathbb{R}^{n_\theta}$ and h a positive smoothing factor (bandwidth).

Step 2.b (state filtering, eq. (9))

We update the state predictive density by using the information coming from $\tilde{\mathbf{y}}_{t+1}^j$ and $\mathbf{y}_{t+1}$, that is

$$p_{N,j}(\mathbf{z}_{t+1}|\mathbf{y}_{1:t+1}, \tilde{\mathbf{y}}_{1:t+1}^j) = \sum_{i=1}^{N} \gamma_{t+1}^{i,j} \delta_{\mathbf{z}_{t+1}^{i,j}}(\mathbf{z}_{t+1}) \tag{15}$$

where $\gamma_{t+1}^{i,j} \propto \omega_t^{i,j} p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}^{i,j}, \tilde{\mathbf{y}}_{t+1}^j)$ is a set of normalized weights and $\mathbf{z}_{t+1}^{i,j} = (\mathbf{x}_{t+1}^{i,j}, \boldsymbol{\theta}_{t+1}^{i,j})$ with $\boldsymbol{\theta}_{t+1}^{i,j} \sim K_h(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t^{i,j})$ and $\mathbf{x}_{t+1}^{i,j} \sim p(\mathbf{x}_{t+1}|\mathbf{x}_t, \boldsymbol{\theta}_{t+1}^{i,j}, \mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t}^j)$, $i = 1, \ldots, N$.

Step 2.c (observation prediction, eq.* (12)*)

The hidden state predictive density can be used to approximate the observable predictive density as follows

$$p_{N,j}(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}, \tilde{\mathbf{y}}_{1:t+1}^j) = \sum_{i=1}^{N} \gamma_{t+1}^{i,j} \delta_{\mathbf{y}_{t+1}^{i,j}}(\mathbf{y}_{t+1}) \tag{16}$$

where $\mathbf{y}_{t+1}^{i,j}$ has been simulated from the combination model $p(\mathbf{y}_{t+1}|\mathbf{z}_{t+1}^{i,j}, \tilde{\mathbf{y}}_{t+1}^j)$ independently for $i = 1, \ldots, N$.

Step 2.d (observation predictive distribution,* (12)*)

The resampling (multinomial resampling) of the particles introduces extra Monte Carlo variations, see Liu and Chen (1998). This can be reduced by doing resampling only when the effective sample size (ESS) is below a given threshold. The ESS is defined as

$$\mathrm{ESS}_t^j = \frac{N}{1 + N \sum_{i=1}^{N} \left(\gamma_{t+1}^{i,j} - N^{-1} \sum_{i=1}^{N} \gamma_{t+1}^{i,j} \right)^2 \Big/ \left(\sum_{i=1}^{N} \gamma_{t+1}^{i,j} \right)^2}.$$

and measures the overall efficiency of an importance sampling algorithm. At the $t+1$-th iteration if $\mathrm{ESS}_{t+1}^j < \kappa$, simulate $\Xi_{t+1}^j = \{\mathbf{z}_{t+1}^{k_i,j}, \omega_{t+1}^{i,j}\}_{i=1}^N$ from $\{\mathbf{z}_{t+1}^{i,j}, \gamma_{t+1}^{i,j}\}_{i=1}^N$ (e.g., multinomial resampling) and set $\omega_{t+1}^{i,j} = 1/N$. We denote with k_i the index of the i-th re-sampled particle in the original set Ξ_{t+1}^j. If $\mathrm{ESS}_{t+1}^j \geq \kappa$ set $\Xi_{t+1}^j = \{\mathbf{z}_{t+1}^{i,j}, \omega_{t+1}^{i,j}\}_{i=1}^N$.

Step 3 (observation marginal predictive, eq.* (10)*)

At the last step, obtain the following empirical predictive density

$$p_{M,N}(\mathbf{y}_{t+1}|\mathbf{y}_{1:t}) = \frac{1}{MN} \sum_{j=1}^{M} \sum_{i=1}^{N} \omega_t^{i,j} \delta_{\mathbf{y}_{t+1}^{i,j}}(\mathbf{y}_{t+1}) \tag{17}$$

The precision of our numerical approximation method depends on the choice of M and N. As regards the precision of the combination weight estimates, it can be increased by setting a larger N. The approximation of the forecasting densities can be increased choosing a larger M. Notice that the choice of M does not affect the convergence of the particle filter algorithms, which depends only on the choice of N. The assumptions required for convergence of the SMC algorithms are discussed in e.g. Del Moral and Guionnet (2001) and Le Gland and Oudjane (2004) for regularized particle filters. Central limit theorems for particle filters can be found in Chan and Lai (2013) and Chopin (2004).

3. Parallel SMC for density combination: DeCo

MATLAB is a popular software in the economics and econometrics community (e.g., see LeSage (1998)), which has recently introduced support to GPU computing in its parallel computing toolbox.

This allows the use of raw CUDA code within a MATLAB program as well as already built-in functions that are directly executed on the GPU. Using the built-in functions we show that GPGPU can be almost effortless where the only knowledge required is a decent MATLAB programming skills. With a little effort we provide GPU implementation of the methodology recently proposed by BCRVD (2013). This implementation provides a speed up of the execution time of up to a hundred times on a multicore CPU with a standard MATLAB code.

3.1. GPU computing in MATLAB

There is little difference between the CPU and GPU MATLAB code: Listings 1 and 2 report the same program which generates and inverts a matrix on CPU and GPU respectively.

```
1 iRows = 1000; iColumns = 1000;% Number of rows and columns
2 C_on_CPU =  randn(iRows, iColumns);% Generate Random number on the CPU
3 InvC_on_CPU = inv(C_on_CPU);% Invert the matrix
```

Listing 1: MATLAB CPU code that generates and inverts a matrix.

```
1 iRows = 1000; iColumns = 1000;% Number of rows and columns
2 C_on_GPU =  gpuArray.randn(iRows, iColumns); % Generate Random number on the
    GPU
3 InvC_on_GPU = inv(C_on_GPU);% Invert the matrix
4 InvC_on_CPU = gather(InvC_on_GPU);% Transfer the data from the GPU to CPU
```

Listing 2: MATLAB GPU code that generate and invert a matrix.

The GPU code, Listing 2, uses the command `gpuArray.randn` to generate a matrix of normal random numbers. The build-in function `gpuArray.randn` is handled by the NVIDIA plug-in that generates the random number with an underlying raw CUDA code. Once the variable `C_on_GPU` is created, standard functions such as `inv` recognize that the variable is on GPU memory and execute the corresponding GPU function, e.g., `inv` is executed directly on the GPU. This is completely transparent to the user. If further calculations are needed on the CPU, then the command `gather` transfers the data from GPU to the CPU, see line 3 of Listing 2. A lot of supported functions already exist and this number is continuously increasing with the new MATLAB releases.

3.2. Parallel sequential Monte Carlo

The structure of the GPU program, which is similar to the CPU one, is reported in Appendix A . In a graphical context the majority of the computations are executed in a single precision floating point, so GPUs were initially optimized to perform these types of computations. Lately, GPUs have been extended to double precision calculation, see Section 4. Since a GPU performs a relatively small set of operations on a specific set of data points (each vertex on the screen), GPU makers (e.g. NVIDIA and ATI) focus mainly on creating hardware that specializes in these tasks instead of a wide array of operations such as the CPU. Therefore, the set of problems in which the GPU can be used is restricted, but GPU performs the specialized tasks more efficiently than the CPU. For example, matrix multiplication and, in general, matrix linear algebra are highly parallelizable problems and these operations are very suitable for GPGPU computing because they can be easily divided into the large number of cores available on the GPU, see Gregory and Miller (2012) for an introduction.

At first sight our problem does not seem to be easily parallelizable. But a closer look shows that the only sequential part of the algorithm is the time iteration, indeed the results of time $t+1$ are dependent

on the results of time t. Our key idea is to rewrite in matrix form that part of the algorithm that iterates over particles and predictive draws in order to exploit the GPGPU computational efficiency. Following the notation in Section 2, we let M be the number of draws from the predictive densities, K the number of predictive models, L the number of variables to predict, T the time horizon, and N the number of particles. Consider $L = 1$ for the sake of simplicity, then the code carries out a matrix of dimension $(MN \times K)$. The dimension could be large, e.g., in our simulation and empirical experiments it is $(5,000 \cdot 1,000 \times 3)$ and $(1,000 \cdot 1,000 \times 12)$ respectively. All the operations, such as addition and multiplication, become just matrix operations that the GPU is explicitly designed for.

As an example of such a coding strategy, we describe the parallel version of the initialization step of the SMC algorithm (see first step of the diagram in Appendix A and subsection (Step 0) in Section 2). We apply a linear regression and then generate a set of normal random numbers to set the initial values of the states. Using a multivariate approach to the regression problem, we can perform it in just one single, big matrix multiplication and inversion for all draws. An example of initialization, similar to the one used in the package, is given in Listing 3.[3]

```
1  %% Initializzation of the matrix %%
2  mX = []; mY = [];
3  for j=1:M
4      %% Build the block-diagonal matrix over the draws   %%
5      mX = blkdiag(mX, mXAll(:, :, j));
6      %% Repat the vY matrix for multivariate regression %%
7      mY = blkdiag(mY, vY);
8  end
9  %% Load data on the GPU memory %%
10 mXGPU = gpuArray(mX);
11 mYGPU = gpuArray(mY);
12 %% Initialize particles on the GPU %%
13 mOmega = (mXGPU\mYGPU); %% Multivariate regression %%
14 mMatrix = kron(gpuArray.ones(M, 1), gpuArray.eye(K));
15 mOmega = mOmega' * mMatrix;
16 mOmega = kron(gpuArray.ones(N, 1), mOmega) + 5 * gpuArray.randn(N * M, K);
```

Listing 3: Block Regression on the GPU.

Listing 3 shows that the predictive densities and the observable variables are stacked in block-diagonal matrices (lines 1-8) of dimensions $(TM \times MK)$ and $(TM \times M)$ by using the command `blkdiag`, and then transferred to the GPU by the `gpuArray` command (lines 10-11). Line 13 carries out the multivariate regression, the function `gpuArray.randn` is then used to generate normal random numbers on the GPU and thus to initialize the value of the particles (lines 14-17).

This strategy is carried out all over the program and also applied to the simulation of the set of particles. For example, listing 4 reports a sample of the code for the SMC prediction step (see subsection (Step 2.a) in Section 2) for the latent states.

```
1  mOmega = mOmega' + kron(gpuArray.ones(1, M * N), sqrt(Sigma)).* gpuArray.
       randn(K, M * N);
```

Listing 4: Draws for the latent states.

The Kronecker product (function `kron`) creates a suitable matrix of standard deviations. We notice that the matrix implementation of the filter requires availability of physical memory on the graphics

[3] In the package, we use the following labelling: **Setting.iDimOmega** for K, **Setting.iDraws** for M and **Setting.cN** for N.

card. If there is not enough memory to run all the draws in parallel, then it is possible to split the M draws in $k = \frac{M}{m}$ blocks of size m and to run the combination algorithm sequentially over the blocks and in parallel within the blocks.[4]

The only step of the algorithm which uses the CPU is resampling (see diagram in Appendix A and subsection (Step 2.d) in Section 2). The generated particles are copied to the CPU memory and, after the necessary calculations, they are passed back to the GPU. Some comments are in order. We use the resampling on CPU because is not easily parallelizable. Although propagation and weighing steps are easy to parallelize (we work on different particles separately), the resampling step requires collective operations across particle weights and those operations are not easily to parallelize. Parallelizing the resampling step with a new and more efficient algorithm is an active field of research, see among others Murray, Lee, and Jacob (2014) and the reference therein, but this is beyond the scope of this paper. Moreover, this copying back and forth introduces a relatively high computational cost in small problems, but becomes much less important as the number of particles and series increases.

4. Differences between CPU and GPU

GPUs can execute calculations in single and double precision as defined by the IEEE 754 standard, IEEE (2008). Single precision numbers are half the size of double precision and they are more limited in the range of values represented. If an application requires a high degree of precision, double precision numbers are the only possibility. We work with double precision numbers because our applications focus on density forecasting and precise estimates of statistical quantities, such as extreme events that are in the tails of the predictive distribution, may be very important for economic and financial decisions.

The GPU cards are very fast in single precision calculation, but lose power in double precision. Therefore, some parameters should be set carefully to have a fair comparison between CPU and GPU. First, both programs have to be implemented in double precision. Second, the CPU program has to be parallel in order to use all available CPU cores. Third, the choice of the hardware is crucial, see Aldrich (2013) for a discussion. In all our experiments, we use a recent CPU processor, Intel Core i7-3820QM, launched in 2012Q2. This CPU has four physical cores that doubled thanks to Hyper-Threading Technology. Not all users of **DeCo** might have access to such up-to-date hardware given its costs. So we also run the CPU code using a less expensive machine, the Intel Xeon X3430, launched in 2009Q3. To run the CPU code in parallel, MATLAB requires the Parallel Computing Toolbox. We also investigate performance when this option is switched off and the CPU code is run sequentially.

The GPU used in this study is a NVIDIA Quadro K2000M. The card is available at a low cost, but it also has low performance because it is designed for a mobile machine (as indicated by the suffix M). A user with a desktop computer might have access to a more powerful video card, such as, e.g., NVIDIA Tesla. We refer to MATLAB for GPU comparisons and, in particular, to the function "GPUBench" freely available at this address `http://www.mathworks.co.uk/matlabcentral/fileexchange/34080-gpubench`.

Finally we emphasize that the results of the CPU and GPU versions of our combination scheme are not necessarily identical. The parallelization is implemented differently in the two versions to fully exploit the GPU advantage in working with a large matrix, selected in the toolbox by the number of

[4]We run the blocks sequentially because MATLAB does not yet have a parallel for loop command for running in parallel the $k = \frac{M}{m}$ blocks of GPU computations. The **DeCo** parallel CPU version sets $m = 1$ and parallelizes over the $k = M$ blocks.

blocks of draws. The CPU is parallelized for each simulation. To investigate the numerical differences between CPU and GPU, we provide a numerical integration experiment based on standard Monte Carlo integration where we can compare numerical solutions when the random generator is fixed and when it is not. We also repeat some of the simulation exercises in BCRVD (2013).

4.1. Monte Carlo integration

We consider six simple integration problems and compare their analytical solutions to their crude Monte Carlo (see Robert and Casella 2004) numerical solutions. Let us consider the two integrals of the function f over the unit interval

$$\mu(f) = \int_0^1 f(x)dx, \quad \sigma^2(f) = \int_0^1 (f(x) - \mu(f))^2 dx.$$

The Monte Carlo approximations of the integrals are

$$\hat{\mu}_N(f) = \frac{1}{N}\sum_{i=1}^{N} f(X_i), \quad \hat{\sigma}^2_N(f) = \frac{1}{N}\sum_{i=1}^{N} (f(X_i) - \hat{\mu}_N(f))^2$$

where $X_1, \ldots, X_N$ is a sequence of N i.i.d. samples from a standard uniform distribution.

The numerical integration problems considered in the experiments correspond to the following choices of the integrand function:

1. $f(x) = x$;
2. $f(x) = x^2$;
3. $f(x) = \cos(\pi x)$.

We repeat $G = 1000$ times each Monte Carlo integration experiment with sample sizes $N = 1500$. The Figures show the CPU and GPU cumulative distribution functions (cdfs) of the mean square errors when the random generator number is fixed equal for the two methods and when it is not. The two cdfs are basically identical when the random generator is fixed, with a difference at the 17th decimal. The differences are larger when the seed is not fixed, but still very small. Moreover, positive and negative values are equally distributed indicating that there is no evidence of higher precision of one of the two methods. We also test the statistical relevance of the differences between CPU and GPU and run a two-sample Kolmogorov-Smirnov test on the cdf of the CPU and GPU squared errors. The results of the tests show that the null hypothesis that CPU and GPU squared errors come from the same distribution cannot be rejected. Thus, we conclude that CPU and GPU also give equivalent results from a statistical point of view even when the random generator number is not set at the same value in the CPU and GPU algorithm. This leads us to move to the simulation exercises in BCRVD(2013).

4.2. Simulation exercises

Following BCRVD (2013) we compare the cases of unbiased and biased predictors and of complete and incomplete model sets using the **DeCo** code. We assume the true model is $\mathcal{M}_1 : y_{1,t} = 0.1 + 0.6 y_{1,t-1} + \varepsilon_{1,t}$ with $\varepsilon_{1,t} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $y_{1,0} = 0.25$, $\sigma = 0.05$ and consider four

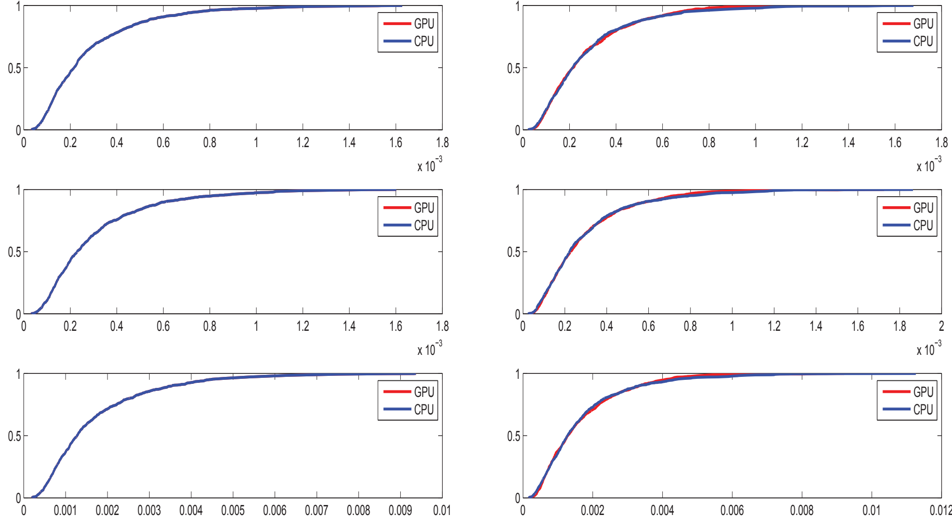

Figure 1: CDFs of the CPU and GPU mean square errors in the MC estimators of $\mu_N(f)$, $N = 1500$, when setting the same random generator number (**left column**) and when not (**right column**), for different choices of f (first rows: $f(x) = x$; second row: $f(x) = x^2$; third row:$f(x) = \cos(\pi x)$), using $G = 1000$ replications.

experiments. We apply the **DeCo** package and use the GUI described in Appendix B to provide the inputs to the combination procedure. We follow BCRVD (2013) and assume that $\boldsymbol{\theta}$ are given.[5]

Complete model set experiments

We assume the true model belongs to the set of models in the combination. In the first experiments the model set also includes two biased predictors: $\mathcal{M}_2 : y_{2,t} = 0.3 + 0.2 y_{2,t-2} + \varepsilon_{2,t}$ and $\mathcal{M}_3 : y_{3,t} = 0.5 + 0.1 y_{3,t-1} + \varepsilon_{3t}$, with $\varepsilon_{it} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$. In the second experiment the complete model set also includes two unbiased predictors: $\mathcal{M}_2 : y_{2,t} = 0.125 + 0.5 y_{2,t-2} + \varepsilon_{2,t}$ and $\mathcal{M}_3 : y_{3,t} = 0.2 + 0.2 y_{3,t-1} + \varepsilon_{3,t}$, with $\varepsilon_{i,t} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$.

Incomplete model set experiments

We assume the true model is not in the model set. In the third experiment the model set includes two biased predictors: $\mathcal{M}_2 : y_{2,t} = 0.3 + 0.2 y_{2,t-2} + \varepsilon_{2,t}$ and $\mathcal{M}_3 : y_{3,t} = 0.5 + 0.1 y_{3,t-1} + \varepsilon_{3,t}$, with $\varepsilon_{i,t} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$. In the fourth experiment the model set includes unbiased predictors: $\mathcal{M}_2 : y_{2,t} = 0.125 + 0.5 y_{2,t-2} + \varepsilon_{2,t}$, $\mathcal{M}_3 : y_{3,t} = 0.2 + 0.2 y_{3,t-1} + \varepsilon_{3,t}$, with $\varepsilon_{i,t} \overset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2)$, $t = 1, \ldots, T$, $i = 2, 3$.

We develop the comparison experiments with both 1000 and 5000 particles. Table 2 reports the time comparison (in seconds) to produce forecast combinations for different experiments and different implementations. Parallel implementation on GPU NVIDIA Quadro K2000M is the most efficient, in

[5]We use the exact same values for the various parameters as in BCRVD (2013) which are not necessarily the same as the default values in the toolbox. See BCRVD (2013) and replication files for further details.

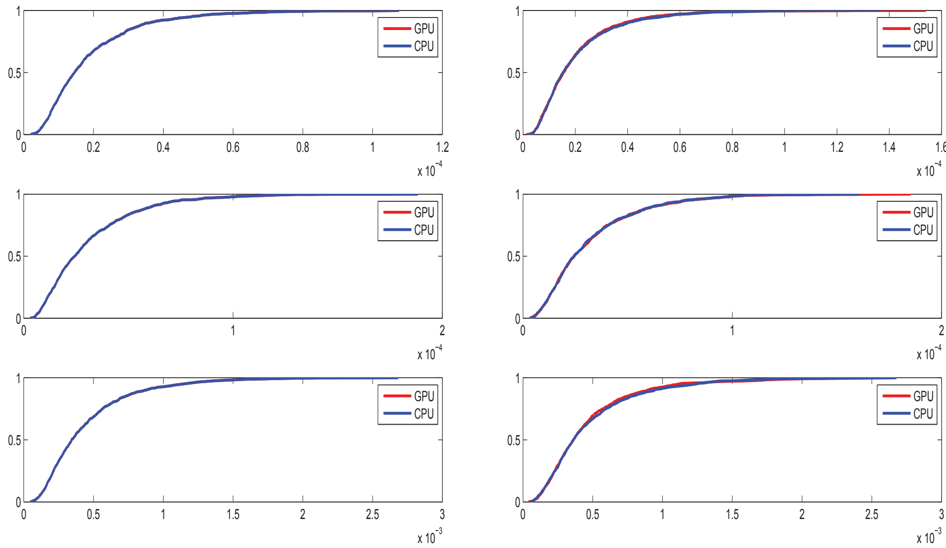

Figure 2: CDFs of the CPU and GPU mean square errors in the MC estimators of $\hat{\sigma}^2_N(f)$ when setting the same random generator number (left column) and when not (right column), for different choices of f (different rows), using $G = 1000$ replications.

terms of computing time, for all experiments. Time differences between the CPU and GPU executions are very large (see Table 2, panel (a)), and result in a saving of up to several hours when using 5000 particles (see Table 2, panel (b)). More specifically, the computational gain of the GPU implementation over parallel CPU implementation varies from 3 to 4 times for the Intel Core i7 and from 5 to 7 times for the Intel Xeon X3430. The overperformance of the parallel GPU implementation on sequential CPU implementation varies from 15 to 20 times when considering an Intel Xeon X3430 machine as a benchmark.

Figure 3 compares the weights for experiments 1 and 2. The weights follow a very similar pattern, but there are some minor discrepancies between them for some observations. Differences are larger for the median value than for the smaller and larger quantiles. The differences are, however, smaller and almost vanish when one focuses on the predictive densities in Figure 4, which is the most important output of the density combination algorithm. We interpret the results as evidence of no economic and statistic significance of the differences between CPU and GPU draws.

The results are similar when focusing on the incomplete model set in Figures 5-6. The evidence does not change when we use 5000 particles.

Learning mechanism experiments

BCRVD (2013) document that a learning mechanism in the weights is crucial to identify the true model (in the case of complete model set) or the best model (in the case of incomplete model set) when the predictions are unbiased, see also left panels in Figures 3-5. We repeat the two unbiased predictor experiments and introduce learning in the combination weights as discussed in Section 2. We set the learning parameters $\lambda = 0.95$ and $\tau = 9$. Tables 3 reports the time comparison (in seconds) when using 1000 and 5000 particles filtered model probability weights. The computation time for **DeCo** increases when learning mechanisms are applied, in particular for the CPU. The GPU

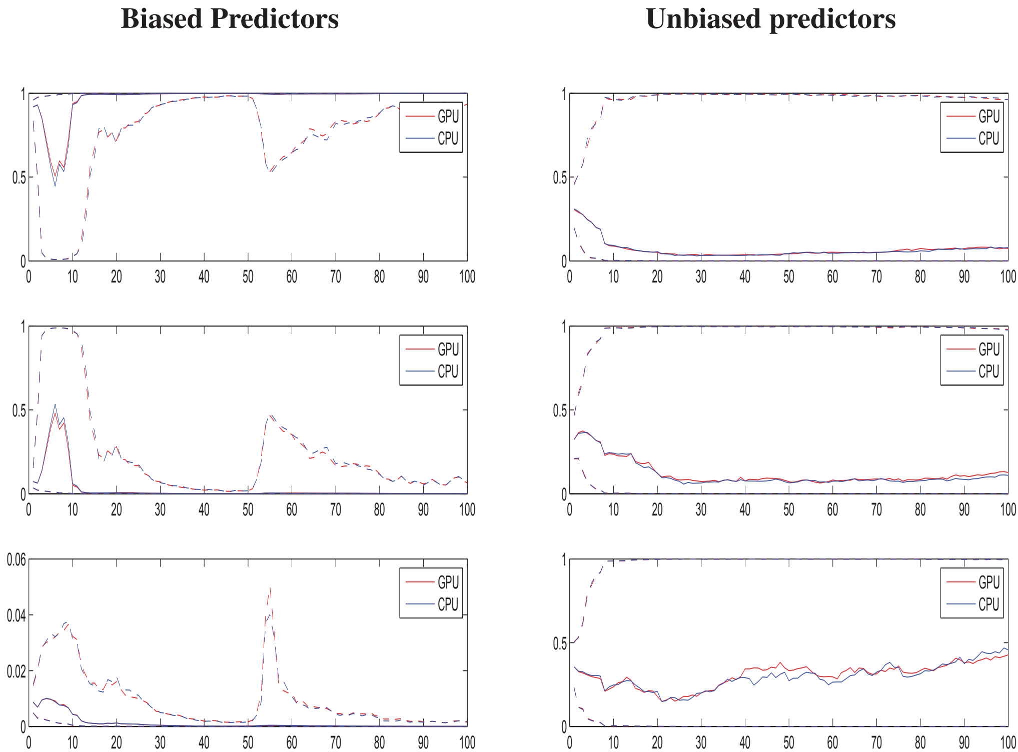

Figure 3: GPU and CPU 1000 particles filtered model probability weights for the complete model set. Median and 95% credibility region for model weights 1,2 and 3 (different rows).

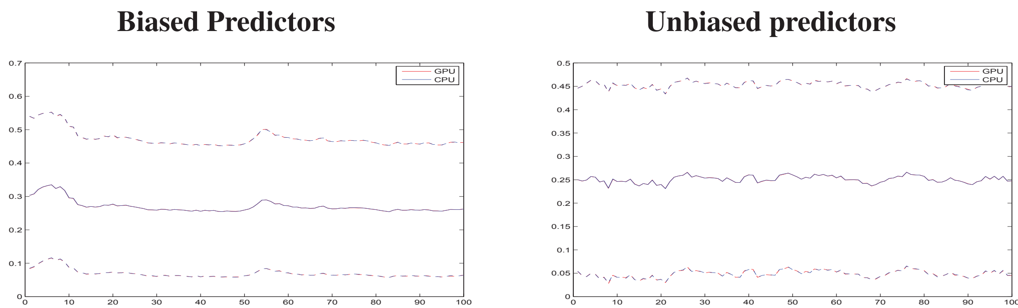

Figure 4: GPU and CPU 1000 particles filtered density forecasts for the complete model set. Mean and 95% credibility region of the combined predictive density.

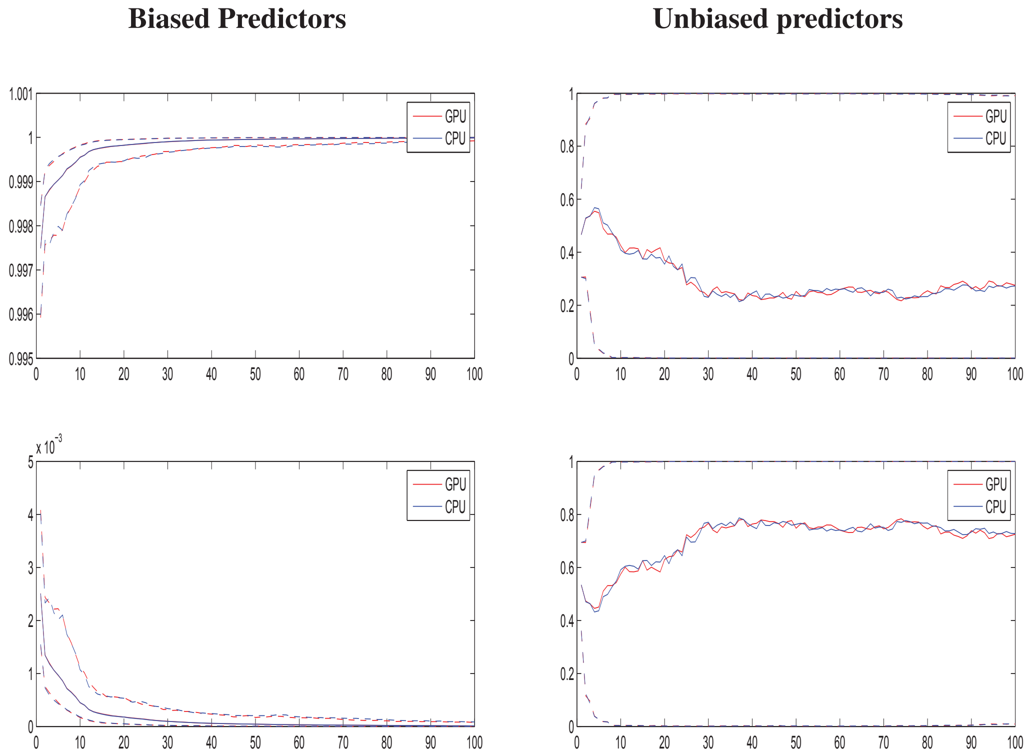

Figure 5: GPU and CPU 1000 particles filtered model probability weights for the incomplete model set. Median and 95% credibility region for model weights 1,2 and 3 (different rows).

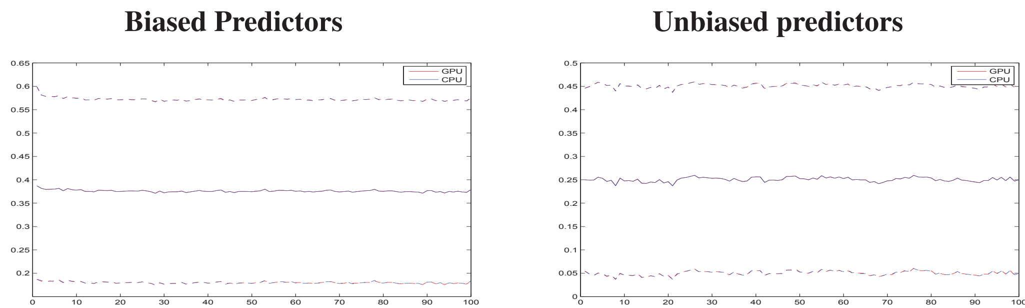

Figure 6: GPU and CPU 1000 particles filtered density forecasts for the incomplete model set. Mean and 95% credibility region of the combined predictive densities.

(a) 1000 Particles

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
		Complete Model Set		
Biased Predictors	699	2780	5119	11749
		(3.97)	(7.32)	(16.80)
Unbiased Predictors	660	2047	5113	11767
		(3.10)	(7.75)	(17.83)
		Incomplete Model Set		
Biased Predictors	671	2801	5112	11635
		(4.17)	(7.62)	(17.34)
Unbiased Predictors	687	2035	5098	11636
		(2.96)	(7.42)	(16.94)

(b) 5000 particles

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
		Complete Model Set		
Biased Predictors	4815	15154	26833	64223
		(3.15)	(5.57)	(13.34)
Unbiased Predictors	5302	15154	26680	63602
		(2.86)	(5.03)	(12.00)
		Incomplete Model Set		
Biased Predictors	4339	13338	26778	64322
		(3.07)	(6.17)	(14.82)
Unbiased Predictors	4581	13203	26762	63602
		(2.88)	(5.84)	(13.88)

Table 2: Density combination computing time in seconds. Rows: different simulation experiments. Columns: parallel GPU (p-GPU) and parallel CPU (p-CPU-i7) implementations on GPU NVIDIA Quadro K2000M with CPU Intel Core i7-3820QM, 3.7GHz; parallel CPU (p-CPU-Xeon) and sequential CPU (CPU-Xeon) implementations on Intel Xeon X3430 4core, 2.40GHz. In parentheses: efficiency gain in terms of CPU/GPU times ratio.

is 10% to 50% slower than without learning, but CPU is 2.5 to almost 4 times slower than previously. The GPU/CPU ratio, therefore, increases in favor of GPU with GPU computation 5 to 70 times faster depending on the alternative CPU machine considered. The **DeCo** codes with learning have some `if` commands related to the minimum numbers of observations necessary to initiate the learning, which increases computational time substantially. The parallelization in GPU is more efficient because it is carried out on blocks of draws and these `if` commands play a minor role. We expect that the gain might increase to several hundreds of times when using parallelization on GPU clusters.

5. Empirical application

As a further check of the performance of the **DeCo** code, we compare the CPU and GPU versions in the macroeconomic application developed in BCRVD (2013). We consider $K = 6$ time series models to predict US GDP growth and PCE inflation: a univariate autoregressive model of order

	p-GPU	p-CPU-i7	p-CPU-Xeon	CPU-Xeon
(a) 1000 Particles				
Complete Model Set	755	7036	14779	52647
		(9.32)	(19.57)	(69.73)
Incomplete Model Set	719	6992	14741	52575
		(9.72)	(20.49)	(73.08)
(b) 5000 particles				
Complete Model Set	7403	35472	73402	274220
		(4.79)	(9.92)	(37.04)
Incomplete Model Set	7260	35292	73256	274301
		(4.86)	(10.09)	(37.78)

Table 3: Density combination computing time in seconds. Rows: different simulation experiments with unbiased predictors and a learning mechanism in the weights. Columns: parallel GPU (p-GPU) and parallel CPU (p-CPU-i7) implementations on GPU NVIDIA Quadro K2000M with CPU Intel Core i7-3820QM, 3.7GHz; parallel CPU (p-CPU-Xeon) and sequential CPU (CPU-Xeon) implementations on Intel Xeon X3430 4core, 2.40GHz. In parentheses: efficiency gain in terms of CPU/GPU times ratio.

one (AR); a bivariate vector autoregressive model for GDP and PCE of order one (VAR); a two-state Markov-switching autoregressive model of order one (ARMS); a two-state Markov-switching vector autoregressive model of order one for GDP and inflation (VARMS); a time-varying autoregressive model with stochastic volatility (TVPARSV); and a time-varying vector autoregressive model with stochastic volatility (TVPVARSV). Therefore, the model set includes constant parameter univariate and multivariate specification; univariate and multivariate models with discrete breaks (Markov-Switching specifications); and univariate and multivariate models with continuous breaks. These are typical models applied in macroeconomic forecasting; see, for example, Clark and Ravazzolo (2012), Korobilis (2013) and D'Agostino, Gambetti, and Giannone (2013).

We evaluate the two combination methods by applying the following evaluation metrics: Root Mean Square Prediction Errors (RMSPE), Kullback Leibler Information Criterion (KLIC) based measure, the expected difference in the Logarithmic Scores (LS) and the Continuous Rank Probability Score (CRPS). Accuracy statistics and related tests (see BCRVD (2013)) are used to compare the forecast accuracy.

Table 4 reports results for the multivariate combination approach. For the sake of brevity, we just present results using parallel GPU and the best parallel CPU Intel Core i7-3820QM machine. We also do not consider a learning mechanism in the weights. GPU is substantially faster, almost 5.5 times faster than CPU, reducing the computational time by more than 5000 seconds. GPU therefore performs relatively better in this experiment than in the previous simulation experiments (without learning mechanisms). The explanation relies on the larger set of models and the multivariate application. The number of simulations has increased substantially and CPU starts to hit physical limits, slowing down the computation and extending time. GPU has no binding limits and just double the time of simulation experiments with a univariate series and the same number of draws and particles.[6] This suggests that GPU might be an efficient methodology to investigate when averaging large sets of

[6] Unreported results show that GPU is more than 36 times faster than sequential CPU implementation on Intel Xeon X3430 4core.

	GDP		Inflation	
	GPU	CPU	GPU	CPU
Time	1249	6923	-	-
RMSPE	0.634	0.637	0.255	0.256
CW	0.000	0.000	0.000	0.000
LS	-1.126	-1.130	0.251	0.257
p-value	0.006	0.005	0.021	0.022
CRPS	0.312	0.313	0.112	0.112
p-value	0.000	0.000	0.000	0.000

Table 4: Computing time and forecast accuracy for the macro-economic application for the GPU (column GPU) and CPU (column CPU) implementations. Rows: Time: time to run the experiment in seconds; RMSPE: Root Mean Square Prediction Error; CW: p-value of the Clark and West (2007) test; LS: average Logarithmic Score over the evaluation period; CRPS: cumulative rank probability score; LS p-value and CRPS p-value: Harvey *et al.* (1997) type of test for LS and CRPS differentials respectively.

models.

Accuracy results for CPU and GPU combinations are very similar and just differ after the third decimal, confirming previous intuitions that the two methods are not necessarily numerically identical, but provide identical economic and statistical conclusions.[7] The combination approach is statistically superior to the AR benchmark for all the three accuracy measures we implement.

6. Conclusion

This paper introduces the MATLAB package **DeCo** (Density Combination) based on parallel Sequential Monte Carlo simulations to combine density forecasts with time-varying weights and different choices of scoring rule.

The package is easy to use for a standard MATLAB user and to facilitate promulgation we have implemented a GUI, which just requires a few input parameters. The package takes full advantage of recent computer hardware progresses and uses banks of parallel SMC algorithms for the density combination using both multi-core CPU and GPU implementation.

The **DeCo** GPU version is up to 70 times faster than the CPU version and even more for larger sets of models. More specifically, our simulation and empirical experiments were conducted using a commercial notebook with CPU Intel Core i7-3820QM and GPU NVIDIA Quadro K2000M, and MATLAB 2013b version, and show that the **DeCo** GPU version is faster than the parallel CPU version, up to 10 times when the weights include a learning mechanism and up to 5.5 times without it, when using an i7 CPU machine and the Parallel Computing Toolbox. These findings are similar to results in Brodtkorb *et al.* (2013) when using a raw CUDA environment. In the comparison between GPU and non-parallel CPU implementations, the differences between GPU and CPU time increase up to almost 70 times when using a standard CPU processor, such as quad-core Xeon. Our results can be further improved with the use of more powerful graphics cards, such as GTX cards. All comparisons have been implemented using double precision for both the CPU and GPU versions. However, if

[7] Numbers for the CPU combination differ marginally (and often just in the third decimal) from those in Table 4 in BCRVD (2013) due to the use of a different MATLAB version, different generator numbers and parallel tooling functions.

an application allows for a lower degree of precision, then single precision calculation can be used and massive gains (up to 500) can be attained, as documented in Lee *et al.* (2010) and Geweke and Durham (2012).

We also document that the CPU and GPU versions do not necessarily provide the exact same numerical solutions to our problems, but differences are not economically and statistically significant. Therefore, users of **DeCo** might choose between the CPU and GPU versions depending on the available and preferred clusters.

Finally, we expect that our research and the **DeCo** GPU implementation would benefit enormously from improvement in the MATLAB parallel computing toolbox, such as the possible incorporation of a parallel "for" loop command for GPU; by the inclusion in the package of different particle filters, such as post- and pre-regularized particle filters and of different density combination schemes; and by applications to large sets of predictive densities as in Casarin, Grassi, Ravazzolo, and van Dijk (2014).

Acknowledgement

For their useful comments, we thank Daniel Armyr and Sylvia Frühwirth-Schnatter and seminar and conference participants at the European Seminar on Bayesian Econometrics 2013, Norges Bank, the 7th Rimini Bayesian Econometrics Workshop and the 2013 Vienna IHS Time-Series Workshop. Roberto Casarin's research is supported by the Italian Ministry of Education, University and Research (MIUR) PRIN 2010-11 grant, and by funding from the European Union, Seventh Framework Programme FP7/2007-2013 under grant agreement SYRTO-SSH-2012-320270. The views expressed in this paper are our own and do not necessarily reflect those of Norges Bank.

References

Aldrich EM (2013). "Massively Parallel Computing in Economics." *Technical report*, University of California, Santa Cruz.

Aldrich EM, Fernández-Villaverde J, Gallant AR, Rubio Ramırez JF (2011). "Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors." *Journal of Economic Dynamics and Control*, **35**, 386–393.

Bates JM, Granger CWJ (1969). "Combination of Forecasts." *Operational Research Quarterly*, **20**, 451–468.

Billio M, Casarin R, Ravazzolo F, van Dijk HK (2013). "Time-varying Combinations of Predictive Densities Using Nonlinear Filtering." *Journal of Econometrics*, **177**, 213–232.

Brodtkorb AR, Hagen TR, Saetra LS (2013). "Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing." *Journal of Parallel and Distributed Computing*, **73**, 4–13.

Casarin R, Grassi S, Ravazzolo F, van Dijk HK (2014). "Dynamic Predictive Density Combinations for Large Datasets." *Unpublished manuscript*, Norges Bank.

Casarin R, Marin JM (2009). "Online Data Processing: Comparison of Bayesian Regularized Particle Filters." *Electronic Journal of Statistics*, **3**, 239 – 258.

Chan H, Lai T (2013). “A general theory of particle filters in hidden Markov models and some applications.” *The Annals of Statistics*, **41**, 2877–2904.

Chong Y, Hendry DF (1986). “Econometric Evaluation of Linear Macroeconomic Models.” *The Review of Economic Studies*, **53**, 671 – 690.

Chopin N (2004). “Central limit theorem for sequential Monte Carlo methods and its application to Bayesian inference.” *The Annals of Statistics*, **32**(6), 2385–2411.

Clark T, Ravazzolo F (2012). “The Macroeconomic Forecasting Performance of Autoregressive Models with Alternative Specifications of Time-Varying Volatility.” *Technical report*, FRB of Cleveland Working Paper 12-18.

Clark T, West K (2007). “Approximately Normal Tests for Equal Predictive Accuracy in Nested Models.” *Journal of Econometrics*, **138**, 291 – 311.

Creal D (2009). “A Survey of Sequential Monte Carlo Methods for Economics and Finance.” *Econometric Reviews*, **31**(3), 245 – 296.

Creel M (2005). “User-Friendly Parallel Computations with Econometric Examples.” *Computational Economics*, **26**, 107 – 128.

Creel M, Goffe WL (2008). “Multi-core CPUs, Clusters, and Grid Computing: A Tutorial.” *Computational Economics*, **32**, 353 – 382.

Creel M, Mandal S, Zubair M (2012). “Econometrics in GPU.” *Technical Report 669*, Barcelona GSE Working Paper.

D’Agostino A, Gambetti L, Giannone D (2013). “Macroeconomic Forecasting and Structural Change.” *Journal of Applied Econometrics*, **28**, 82 – 101.

Del Moral P, Guionnet A (2001). “On the stability of interacting processes with applications to filtering and genetic algorithms.” *Ann. Inst. H. Poincaré Probab. Statist.*, **37**, 155–194.

Doucet A, Freitas JG, Gordon J (2001). *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, New York.

Dziubinski MP, Grassi S (2013). “Heterogeneous Computing in Economics: A Simplified Approach.” *Computational Economics*, pp. 1245 – 1266.

Geweke J, Amisano G (2010). “Optimal Prediction Pools.” *Journal of Econometrics*, **164**(2), 130 – 141.

Geweke J, Durham G (2012). “Massively Parallel Sequential Monte Carlo for Bayesian Inference.” *Working papers*, National Bureau of Economic Research, Inc.

Gneiting T, Raftery AE (2007). “Strictly Proper Scoring Rules, Prediction, and Estimation.” *Journal of the American Statistical Association*, **102**, 359 – 378.

Granger CWJ, Ramanathan R (1984). “Improved Methods of Combining Forecasts.” *Journal of Forecasting*, **3**, 197 – 204.

Gregory K, Miller A (2012). *Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press, USA.

Hall SG, Mitchell J (2007). "Combining Density Forecasts." *International Journal of Forecasting*, **23**, 1 – 13.

Harvey D, Leybourne S, Newbold P (1997). "Testing the Equality of Prediction Mean Squared Errors." *International Journal of Forecasting*, **13**, 281 – 291.

Hoberock J, Bell N (2011). *Thrust: A parallel template library, Version 1.4.0*.

IEEE (2008). *IEEE 754 - 2008. IEEE 754 - 2008 Standard for Floating-Point Arithmetic*. IEEE.

Jore AS, Mitchell J, Vahey SP (2010). "Combining Forecast Densities from VARs with Uncertain Instabilities." *Journal of Applied Econometrics*, **25**(4), 621 – 634.

Khronos OpenCL Working Group (2009). *The OpenCL Specification Version 1.0. Khronos Group*. URL http://www.khronos.org/opencl.

Korobilis D (2013). "VAR Forecasting Using Bayesian Variable Selection." *Journal of Applied Econometrics*, **28**, 204 – 230.

Le Gland F, Oudjane N (2004). "Stability and uniform approximation of nonlinear filters using the Hilbert metric and application to particle filters." *The Annals of Applied Probability*, **14**(1), 144–187.

Lee A, Christopher Y, Giles MB, Doucet A, Holmes CC (2010). "On the Utility of Graphic Cards to Perform Massively Parallel Simulation with Advanced Monte Carlo Methods." *Journal of Computational and Graphical Statistics*, **19**, 769 – 789.

Legland F, Oudjane N (2004). "Stability and Uniform Approximation of Nonlinear Filters using the Hilbert Metric and Application to Particle Filters." *The Annals of Applied Probability*, **14**, 144 – 187.

LeSage JP (1998). "ECONOMETRICS: MATLAB Toolbox of Econometrics Functions." Statistical Software Components, Boston College Department of Economics.

Liu J, Chen R (1998). "Sequential Monte Carlo Methods for Dynamical System." *Journal of the American Statistical Association*, **93**, 1032 – 1044.

Liu JS, West M (2001). "Combined Parameter and State Estimation in Simulation Based Filtering." In A Doucet, N de Freitas, N Gordon (eds.), *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Morozov S, Mathur S (2011). "Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control." *Computational Economics*, pp. 1 – 32.

Murray LM, Lee A, Jacob PE (2014). "Parallel resampling in the particle filter." *Working paper*.

Musso C, Oudjane N, Legland F (2001). "Improving Regularised Particle Filters." In A Doucet, N de Freitas, N Gordon (eds.), *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Nvidia Corporation (2010). *Nvidia CUDA Programming Guide, Version 3.2*. URL http://www.nvidia.com/CUDA.

Press WH, Teukolsky ST, Vetterling WT, Flannery BP (1992). *Numerical Recipes in C: the Art of Scientific Computing*. 2nd edition. Cambridge University Press, Cambridge, England.

Robert CP, Casella G (2004). *Monte Carlo Statistical Methods*. 2nd edition. Springer-Verlag, Berlin.

Stroustrup B (2000). *The C++ Programming Language*. 3rd edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Suchard M, Holmes C, West M (2010). "Some of the What?, Why?, How?, Who? and Where? of Graphics Processing Unit Computing for Bayesian Analysis." *Bulletin of the International Society for Bayesian Analysis*, **17**, 12 – 16.

Sutter H (2005). "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." *Dr. Dobb's Journal*. URL http://www.gotw.ca/publications/concurrencyddj.htm.

Sutter H (2011). "Welcome to the Jungle." URL http://herbsutter.com/welcome-to-the-jungle/.

Swann CA (2002). "Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI." *Computational Economics*, **19**, 145 – 178.

Terui N, van Dijk HK (2002). "Combined Forecasts from Linear and Nonlinear Time Series Models." *International Journal of Forecasting*, **18**, 421 – 438.

The MathWorks, Inc (2011). *MATLAB – The Language of Technical Computing, Version R2011b*. The MathWorks, Inc., Natick, Massachusetts. URL http://www.mathworks.com/products/matlab/.

A . Flow-chart of GPU DeCo package

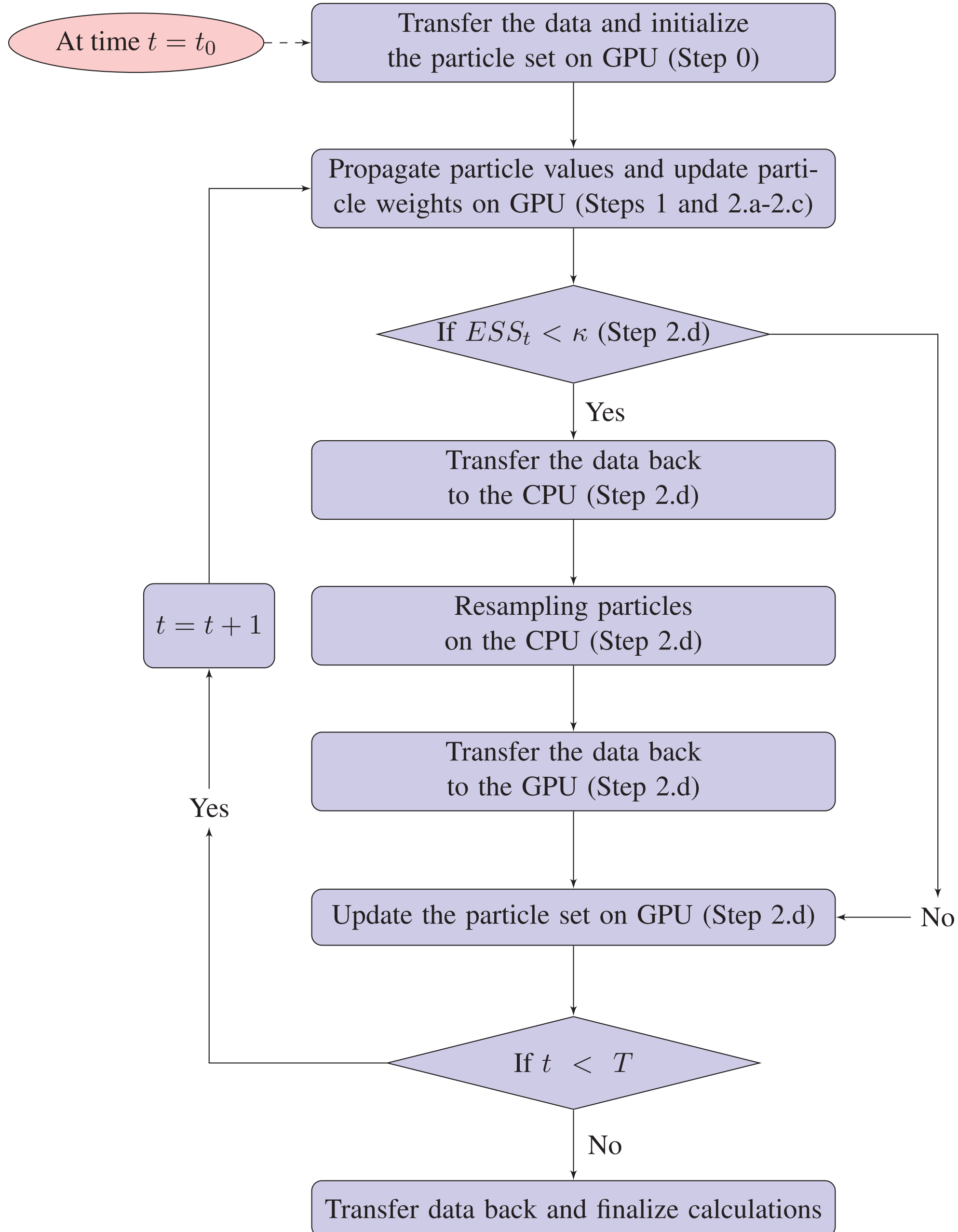

Figure 7: Flow chart of the parallel SMC filter given in Section 2.

B . The GUI

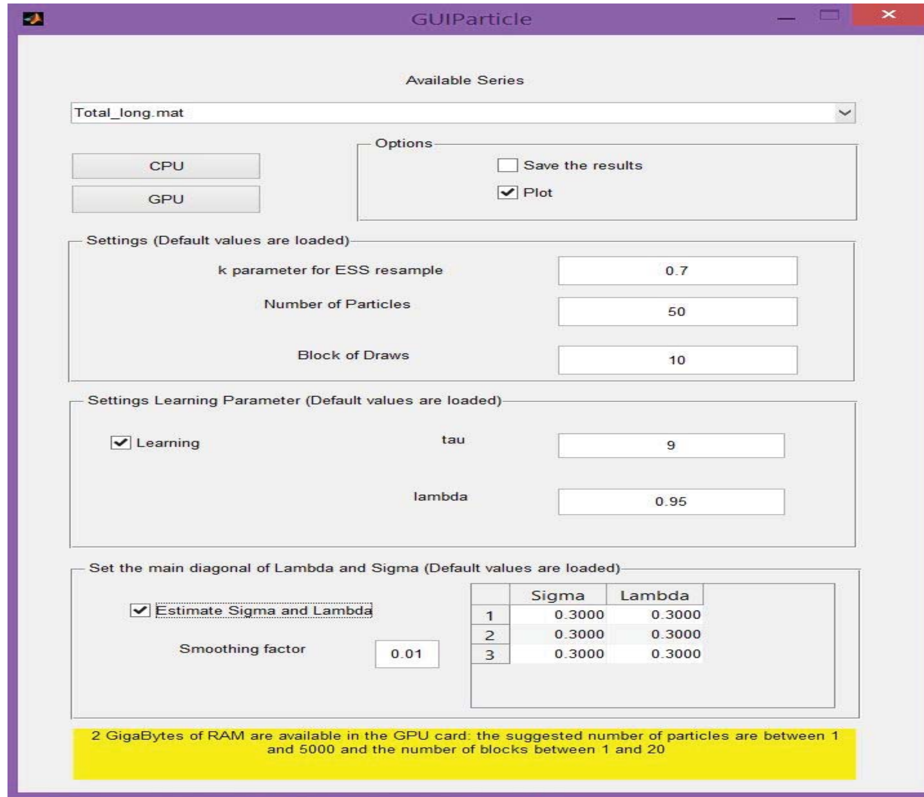

Figure 8: The GUI of the **DeCo** package, case of GPU available.

Figure 8 shows the GUI of the **DeCo** package, which contains all the necessary inputs for our program. The ListBox loads and displays the available dataset in the directory *Dataset*. The figure shows, as example, the dataset `Total_long.mat`. The directory *Dataset* also contains other two dataset of different dimensions. The three databases differ in sample size, number of predictive series and number of series to be predicted.

The `Panel` "Options" contains the command for saving and plotting the results. The results are saved in the directory *OutputCPU* or *OutputGPU* depending on the type of calculation chosen.

The `Panel` "Settings" contains the selection of the number of particles, the number of blocks of draws (see Section 3.2) and the resampling threshold κ. All three options have default values that are also reported. The number of block of draws is only relevant for the GPU version.

The `Panel` "Setting Learning Parameter" allows the user to perform the calculation with or without learning in the weights, see Section 4.2. When the option Learning is chosen, the edit box allows the learning parameter to be set, the default values are $\lambda = 0.95$ and $\tau = 9$.

The `Panel` "Set the main diagonal of Lambda and Sigma" reports an editable table that allows the values for Sigma and Lambda to be set (default values are loaded). The check box "Estimate Sigma and Lambda" allows the estimation of the Sigma and Lambda matrix. Once the option is selected, the editable table reports the mean values of the priors and the user has also to select the variance of the shocks (smoothing factor) for those matrice. A default value of 0.01 is set.

Finally the bottom "CPU" starts the corresponding CPU program and the bottom "GPU" executes the program on the GPU. In the case of no GPU card available, the corresponding bottom and the edit box "blocks of draws" are not visible, see Figure 9 with as example the dataset `Total_medium.mat`.

Some considerations are in order. First, the CPU is already implemented in parallel form, the user has to start a parallel session in MATLAB by typing the command $matlabpool\ open$ in the MATLAB main window. Please refer to MATLAB online help and to the ToolboxDescription. Second, the dataset accepted by the program is `mat` format and has the following structure. It includes two variables, the first one is defined as vY and it contains a $(T \times L)$ matrix of the the variables $\{\mathbf{y}_t\}_{t=1}^{T}$ to be predicted, where T is the number of 1-step ahead forecasts and L the size of observable variables to forecast. The second one is a $4-D$ matrix defined mX with the following dimensions (T, M, L, KL) , where M is the size of i.i.d. samples from the predictive densities, and KL the number of 1-step ahead predictive densities. Finally, the user might apply different learning mechanisms based on other scoring functions that the one applied and discussed in Section 2. In order to do this, the user should change the functions "PFCoreCPU.m" and "PFCoreGPU.m".

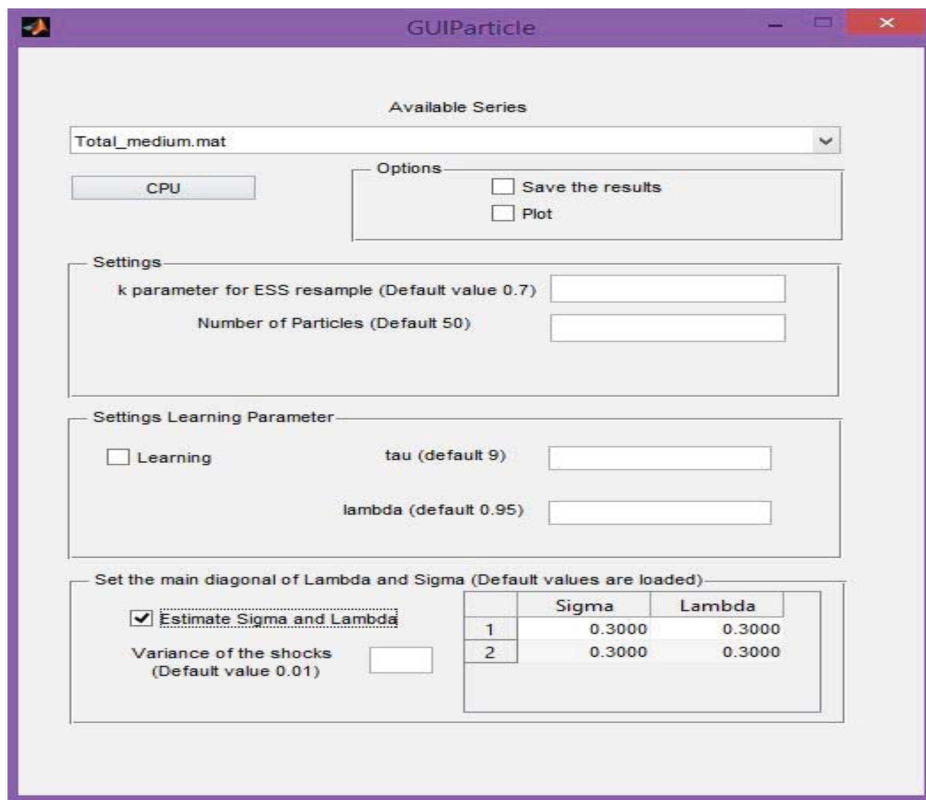

Figure 9: The GUI of the **DeCo** package, case of no GPU available.