View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Apollo

Journal of Statistical Software
October 2020, Volume 95, Issue 7. doi: 10.18637/jss.v095.i07

MultiBUGS: A Parallel Implementation of the BUGS Modeling Framework for Faster Bayesian Inference

Robert J. B. Goudie
University of Cambridge

Rebecca M. Turner
University College London

Daniela De Angelis
University of Cambridge

Andrew Thomas
University of Cambridge

Abstract

MultiBUGS is a new version of the general-purpose Bayesian modeling software BUGS that implements a generic algorithm for parallelizing Markov chain Monte Carlo (MCMC) algorithms to speed up posterior inference of Bayesian models. The algorithm parallelizes evaluation of the product-form likelihoods formed when a parameter has many children in the directed acyclic graph (DAG) representation; and parallelizes sampling of conditionally-independent sets of parameters. A heuristic algorithm is used to decide which approach to use for each parameter and to apportion computation across computational cores. This enables **MultiBUGS** to automatically parallelize the broad range of statistical models that can be fitted using BUGS-language software, making the dramatic speed-ups of modern multi-core computing accessible to applied statisticians, without requiring any experience of parallel programming. We demonstrate the use of **MultiBUGS** on simulated data designed to mimic a hierarchical e-health linked-data study of methadone prescriptions including 425,112 observations and 20,426 random effects. Posterior inference for the e-health model takes several hours in existing software, but **MultiBUGS** can perform inference in only 28 minutes using 48 computational cores.

Keywords: BUGS, parallel computing, Markov chain Monte Carlo, Gibbs sampling, Bayesian analysis, hierarchical models, directed acyclic graph.

1. Introduction

BUGS is a long running project that makes easy to use Bayesian modeling software available to the statistics community. The software has evolved through three main versions since 1989: first **ClassicBUGS** (Spiegelhalter, Thomas, Best, and Gilks 1996), then **WinBUGS** (Lunn,

Thomas, Best, and Spiegelhalter 2000), then the current open-source **OpenBUGS** (Lunn, Spiegelhalter, Thomas, and Best 2009). The software is structured around the twin ideas of the declarative BUGS language (Thomas 2006), through which the user specifies the graphical model (Lauritzen, Dawid, Larsen, and Leimer 1990) that defines the statistical model to be analyzed; and Markov chain Monte Carlo simulation (MCMC; Geman and Geman 1984; Gelfand and Smith 1990), which is used to estimate the posterior distribution. These ideas have also been widely adopted in other Bayesian software, notably in **JAGS** (Plummer 2017) and NIMBLE (de Valpine, Turek, Paciorek, Anderson-Bergman, Lang, and Bodik 2017), and related ideas are used in Stan (Carpenter *et al.* 2017).

Technological advances in recent years have led to massive increases in the amount of data that are generated and stored. This has posed problems for traditional Bayesian modeling, because fitting such models with a huge amount of data in existing standard software, such as **OpenBUGS**, is typically either impossible or extremely time-consuming. While most recent computers have multiple computational cores, which can be used to speed up computation, **OpenBUGS** has not previously made use of this facility. The aim of **MultiBUGS** is to make available to applied statistics practitioners the dramatic speed-ups of multi-core computation without requiring any knowledge of parallel programming, through an easy-to-use implementation of a generic, automatic algorithm for parallelizing the MCMC algorithms used by BUGS-style software.

1.1. Approaches to MCMC parallelization

The most straightforward approach for using multiple computational cores or multiple central processing units (CPUs) to perform MCMC simulation is to run each of multiple, independent MCMC chains on a separate CPU or core (e.g., Bradford and Thomas 1996; Rosenthal 2000). Since the chains are independent, there is no need for information to be passed between the chains: The algorithm is embarrassingly parallel. Running several MCMC chains is valuable for detecting problems of non-convergence of the algorithm using, for example, the Brooks-Gelman-Rubin diagnostic (Gelman and Rubin 1992; Brooks and Gelman 1998). However, the time taken to get past the burn in period cannot be shortened using this approach.

A different approach is to use multiple CPUs or cores for a single MCMC chain, with the aim of shortening the time taken for the MCMC chain to converge and to mix. One way to do this is to identify tasks within standard MCMC algorithms that can be calculated in parallel, without altering the underlying Markov chain. A task that is often, in principle, straightforward to parallelize, and is fundamental in several MCMC algorithms, such as the Metropolis-Hastings algorithm, is evaluation of the likelihood (e.g., Whiley and Wilson 2004; Jewell, Kypraios, Neal, and Roberts 2009; Bottolo *et al.* 2013). Another task that can be parallelized is sampling of conditionally-independent components, as suggested by, for example, Wilkinson (2006).

MultiBUGS implements all of the above strategies for parallelization of MCMC. There are thus two levels of parallelization: Multiple MCMC chains are run in parallel, with the computation required by each chain also parallelized by identifying both complex parallelizable likelihoods and conditionally-independent components that can be sampled in parallel.

There are numerous other approaches to MCMC parallelization. Several authors have proposed running parts of the model on separate cores and then combining results (Scott, Blocker, Bonassi, Chipman, George, and McCulloch 2016) using either somewhat ad hoc procedures or sequential Monte Carlo-inspired methods (Goudie, Presanis, Lunn, De Angelis, and Wernisch

2018). This approach has the advantage of being able to reuse already written MCMC software and, in this sense, is similar to the approach used in **MultiBUGS**. A separate body of work (Brockwell 2006; Angelino, Kohler, Waterland, Seltzer, and Adams 2014) proposes using a modified version of the Metropolis-Hastings algorithm which speculatively considers a possible sequence of MCMC steps and evaluates the likelihood at each proposal on a separate core. The time saving tends to scale logarithmically in the number of cores for this class of algorithms. A final group of approaches modifies the Metropolis-Hastings algorithm by proposing a sequence of candidate points in parallel (Calderhead 2014). This approach can reduce autocorrelations in the MCMC chain and so speed up MCMC convergence.

1.2. MultiBUGS software

MultiBUGS is available as free software, under the GNU General Public License version 3, and can be downloaded from `https://www.multibugs.org/`. **MultiBUGS** currently requires Microsoft Windows, and version 8.1 or newer of the Microsoft MPI (MS-MPI) parallel programming framework[1]. Note that the Windows Firewall may require you to give **MultiBUGS** permission to communicate between cores. The source code for **MultiBUGS** can be downloaded from `https://github.com/MultiBUGS/MultiBUGS`. The data and model files to replicate all the results presented in this paper can be found within **MultiBUGS**, as we describe later in the paper, or can be downloaded from `https://github.com/MultiBUGS/multibugs-examples`.

The paper is organized as follows: In Section 2 we introduce the class of models we consider and the parallelization strategy adopted in **MultiBUGS**; implementation details are provided in Section 3; Section 4 summarizes the basic process of fitting models in **MultiBUGS**; Section 5 demonstrates **MultiBUGS** for analyzing a large hierarchical dataset; and we conclude with a discussion in Section 6.

2. Background and methods

2.1. Models and notation

MultiBUGS performs inference for Bayesian models that can be represented by a directed acyclic graph (DAG), with each component of the model associated with a node in the DAG. A DAG $G = (V_G, E_G)$ consists of a set of nodes or vertices V_G joined by directed edges $E_G \subset V_G \times V_G$, represented by arrows. The parents $\text{pa}_G(v) = \{u : (u, v) \in E_G\}$ of a node v are the nodes with an edge pointing to node v. The children $\text{ch}_G(v) = \{u : (v, u) \in E_G\}$ of a node v are the nodes pointed to by edges emanating from node v. We omit G subscripts here, and throughout the paper, wherever there is no ambiguity.

DAGs can be presented graphically (see Figures 1 and 3), with stochastic nodes shown in ovals, and constant and observed quantities in rectangles. Stochastic dependencies are represented by arrows. Repeated nodes are enclosed by a rounded rectangle (plate), with the range of repetition indicated by the label.

To establish ideas, consider a simple random effects logistic regression model (called "seeds") for the number r_i of seeds that germinated out of n_i planted, in each of $i = 1, \ldots, N = 21$ experiments, with binary indicators of seed type X_{1i} and root extract type X_{2i} (Crowder

[1] Available at `https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx`.

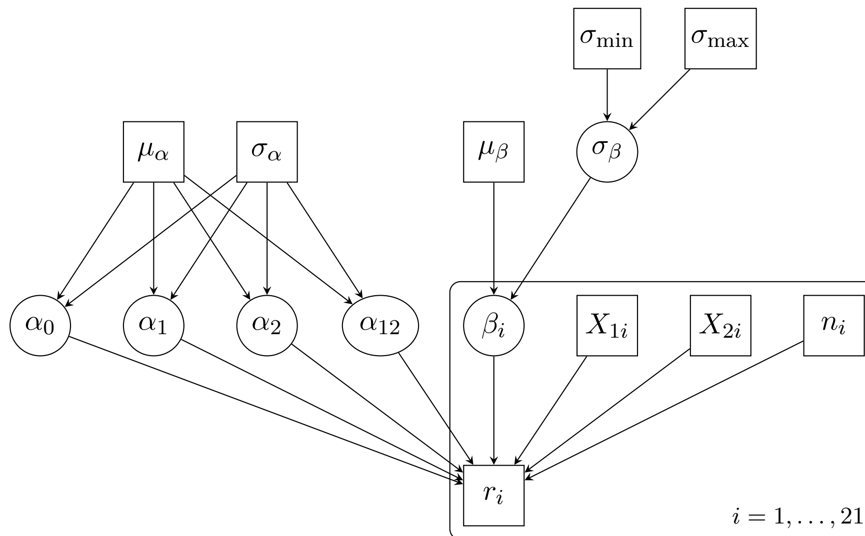

Figure 1: DAG representation of the seeds model.

1978; Breslow and Clayton 1993).

$$r_i \sim \text{Bin}(p_i, n_i)$$
$$\text{logit}(p_i) = \alpha_0 + \alpha_1 X_{1i} + \alpha_2 X_{2i} + \alpha_{12} X_{1i} X_{2i} + \beta_i$$
$$\beta_i \sim \text{N}(\mu_\beta, \sigma_\beta^2)$$

We choose normal priors for the regression parameters $\alpha_0, \alpha_1, \alpha_2, \alpha_{12}$, with mean $\mu_\alpha = 0$ and standard deviation $\sigma_\alpha = 1000$. We fix $\mu_\beta = 0$, and choose a uniform prior on the range $\sigma_{\min} = 0$ to $\sigma_{\max} = 10$ for the standard deviation σ_β of the random effects β_i. Figure 1 shows a DAG representation of the "seeds" model. The data are presented in Crowder (1978).

For ease of exposition of the parallelization methods used by **MultiBUGS**, we assume throughout this paper that the set of nodes V_G includes all stochastic parameters $S_G \subseteq V_G$ and constant quantities (including observations and hyperparameters) in the model, but excludes parameters that are entirely determined by other parameters. As a consequence, the DAG for the seeds example (Figure 1) includes as nodes the stochastic parameters $S_G = \{\alpha_0, \alpha_1, \alpha_2, \alpha_{12}, \beta_1, \ldots, \beta_{21}, \sigma_\beta\}$, the observations $\{r_i, X_{1i}, X_{2i}, n_i : i = 1, \ldots, 21\}$ and the constant hyperparameters $\{\mu_\alpha, \sigma_\alpha, \mu_\beta, \sigma_{\min}, \sigma_{\max}\}$, but not the parameters that are deterministic functions of other parameters (the germination probabilities p_i), which have been assimilated into the definition of the distribution of r_i before forming the DAG. Arbitrary DAG models can nevertheless be considered by assimilating deterministic intermediary quantities, such as linear predictors in generalized linear models, into the definition of the conditional distribution of the appropriate descendant stochastic parameter; and considering deterministic prediction separately from the main MCMC computation. For example, in the seeds example, the random effect precision $\tau_\beta = \sigma_\beta^{-2}$ is deterministically related to the standard deviation σ_β, so it would not be considered part of the graph if it were of interest: Posterior inference for τ_β could instead be made either by updating its value in the usual (serial) manner after each MCMC iteration, or by post-processing the MCMC samples for σ_β.

In DAG models, the conditional independence assumptions represented by the DAG mean that the full joint distribution of all quantities V has a simple factorization in terms of the

conditional distribution $p(v \mid \mathrm{pa}(v))$ of each node $v \in V$ given its parents $\mathrm{pa}(v)$:

$$p(V) = \prod_{v \in V} p(v \mid \mathrm{pa}(v))$$

Posterior inference is performed in **MultiBUGS** by an MCMC algorithm, constructed by associating each node with a suitable updating algorithm, chosen automatically by the program according to the structure of the model. Most MCMC algorithms involve evaluation of the conditional distribution of the stochastic parameters $S \subseteq V$ (at particular values of its arguments). The conditional distribution $p(v \mid V_{-v})$ of a node $v \in S$, given the remaining nodes $V_{-v} = V \setminus \{v\}$ is

$$p(v \mid V_{-v}) \propto p(v \mid \mathrm{pa}(v)) L(v), \tag{1}$$

where $p(v \mid \mathrm{pa}(v))$ is the prior factor and $L(v) = \prod_{u \in \mathrm{ch}(v)} p(u \mid \mathrm{pa}(u))$ is the likelihood factor.

2.2. Parallelization methods in MultiBUGS

MultiBUGS performs in parallel both multiple chains and the computation required for a single MCMC chain. In this section, we describe how the computation for a single MCMC chain can be performed in parallel.

Parallelization strategies

MCMC entails sampling, which often requires evaluation of the conditional distribution of the stochastic parameters S in the model. **MultiBUGS** parallelizes these computations for a single MCMC chain via two distinct approaches.

First, when a parameter has many children, evaluation of the conditional distribution is computationally expensive, since Equation 1 is the product of many terms. However, the evaluation of the likelihood factor $L(v)$ can easily be split between C cores by calculating a partial product involving every Cth child on each core. With a partition $\{\mathrm{ch}^{(1)}(v), \ldots, \mathrm{ch}^{(C)}(v)\}$ of the set of children $\mathrm{ch}(v)$, we can evaluate $\prod_{u \in \mathrm{ch}^{(c)}(v)} p(u \mid \mathrm{pa}(u))$ on the cth core, $c = 1, \ldots, C$. The prior factor $p(v \mid \mathrm{pa}(v))$ and these partial products can be multiplied together to recover the complete conditional distribution.

Second, when a model includes a large number of parameters then computation may be slow in aggregate, even if sampling of each individual parameter is fast. However, parameters can clearly be sampled in parallel if they are conditionally independent. Specifically, all parameters in a set $W \subseteq S$ can be sampled in parallel whenever the parameters in W are mutually conditionally-independent; i.e., all $w_1 \in W$ and $w_2 \in W$ $(w_1 \neq w_2)$ are conditionally independent given $V \setminus W$. If C cores are available and $|W|$ denotes the number of elements in the set W, then in a parallel scheme at most $\lceil |W|/C \rceil$ parameters need to be sampled on a core (where $\lceil x \rceil$ denotes the ceiling function), rather than $|W|$ in the standard serial scheme.

To identify sets of conditionally-independent parameters, **MultiBUGS** first partitions the stochastic parameters S into depth sets $D_G^h = \{v \in S : d_G(v) = h\}$, defined as the set of stochastic nodes with topological depth $d_G(v) = h$, where topological depth of a node $v \in V$ is defined recursively, starting from the nodes with no parents.

$$d_G(v) = \begin{cases} 0 & \text{if } \mathrm{pa}_G(v) = \varnothing \\ 1 + \max_{u \in \mathrm{pa}_G(v)} d_G(u) & \text{otherwise} \end{cases}$$

Note that stochastic nodes $v \in S$ have topological depth $d_G(v) \geq 1$, since the constant hyperparameters of stochastic nodes are included in the DAG.

Sets of conditionally-independent parameters within a depth set can be identified by noting that all parameters in a set $W \subseteq D_G^h$ are mutually conditionally-independent, given the other nodes $V \setminus W$, if the parameters in W have no child node in common. This follows from the d-separation criterion (Definition 1.2.3, Pearl 2009): All such pairs of parameters $w_1 \in W$ and $w_2 \in W$ $(w_1 \neq w_2)$ are d-separated by $V \setminus W$ because no "chain path" can exist between w_1 and w_2 because these nodes have the same topological depth; and all "fork paths" are blocked by $V \setminus W$, as are all "collider paths", except those involving a common child of w_1 and w_2, which are prevented by definition of W.

Heuristic for determining parallelization strategy

A heuristic criterion is used by **MultiBUGS** to decide which type of parallelism to exploit for each parameter in the model. The heuristic aims to parallelize the evaluation of conditional distributions of "fixed effect"-like parameters, and parallelize the sampling of "random effect"-like parameters. The former tend to have a large number of children, whereas the latter tend to have a small number of children. Each depth set is considered in turn, starting with the "deepest" set $D_G^{h^\star}$ with $h^\star = \max_{v \in S} d_G(v)$. The computation of the parameter's conditional distribution is parallelized if a parameter has more children than double the mean number of children $\overline{\mathrm{ch}} = \mathrm{mean}_{v \in S} |\,\mathrm{ch}_G(v)|$, or if all parameters in the graph have topological depth $h = 1$; otherwise the sampling of conditionally independent sets of parameters is parallelized whenever this is permitted. The special case for $h = 1$ ensures that evaluation of the conditional distribution of parameters is parallelized in "flat" models in which all parameters have identical topological depth. When a group of parameters is sampled in parallel we would like the time taken to sample each one to be similar, so **MultiBUGS** assigns parameters to cores in order of the number of children that each parameter has.

MultiBUGS creates a C-column computation schedule table T, which specifies the parallelization scheme: Where different parameters appear in a row, the corresponding parameters are sampled in parallel; where a single parameter is repeated across a full row, the evaluation of the conditional distribution for that parameter is split into partial products across the C cores. A single MCMC iteration consists of evaluating updates as specified by each row of the computation schedule in turn. The computation schedule includes blanks whenever a set W of mutually conditionally-independent parameters does not divide equally across the C cores; that is, when $|W| \bmod C \neq 0$, where mod denotes the modulo operator. The corresponding cores are idle when a blank occurs. Appendix A describes the algorithms used to create the C-column computation schedule table T in detail.

We illustrate the heuristic by describing the process of creating Table 1, the computation schedule for the seeds example introduced in Section 2, assuming $C = 4$ cores are available. The model includes 26 stochastic parameters $S = \{\alpha_0, \alpha_1, \alpha_2, \alpha_{12}, \beta_1, \ldots, \beta_{21}, \sigma_\beta\}$; and $|\,\mathrm{ch}(\alpha_0)| = |\,\mathrm{ch}(\alpha_1)| = |\,\mathrm{ch}(\alpha_2)| = |\,\mathrm{ch}(\alpha_{12})| = |\,\mathrm{ch}(\sigma_\beta)| = 21$ and $|\,\mathrm{ch}(\beta_1)| = \cdots = |\,\mathrm{ch}(\beta_{21})| = 1$. **MultiBUGS** first considers the parameters $\beta_1, \ldots, \beta_{21}$, since the topological depth $d(\beta_1) = \cdots = d(\beta_{21}) = 2 = \max_{v \in S} d(v)$. None of the likelihood evaluation for $\beta_1, \ldots, \beta_{21}$ is parallelized, because all these parameters have only 1 child and $\overline{\mathrm{ch}} \approx 4.8$. However, $\beta_1, \ldots, \beta_{21}$ are mutually conditionally-independent and so these parameters are distributed across the 4 cores as shown in the first 6 rows of Table 1. Since $21 \bmod 4 \neq 0$,

Row	Core 1	2	3	4
1	β_1	β_2	β_3	β_4
2	β_5	β_6	β_7	β_8
3	β_9	β_{10}	β_{11}	β_{12}
4	β_{13}	β_{14}	β_{15}	β_{16}
5	β_{17}	β_{18}	β_{19}	β_{20}
6	β_{21}			
7	α_{12}	α_{12}	α_{12}	α_{12}
8	α_1	α_1	α_1	α_1
9	α_2	α_2	α_2	α_2
10	α_0	α_0	α_0	α_0
11	σ_β	σ_β	σ_β	σ_β

Table 1: Computation schedule table T for the seeds example, with 4 cores.

cores 2, 3 and 4 will be idle while β_{21} is sampled. Next, we consider $\alpha_0, \alpha_1, \alpha_2, \alpha_{12}$ and σ_β, since $d(\alpha_0) = \cdots = d(\alpha_{12}) = d(\sigma_\beta) = 1$. Since all of these parameters have 21 children and $\overline{\mathrm{ch}} \approx 4.8$, **MultiBUGS** will spread the likelihood evaluation of all these parameters across cores, and these are assigned to the computation schedule in turn.

Block samplers

MultiBUGS is able to use a block MCMC sampler when appropriate: that is, algorithms that sample a block of nodes jointly, rather than just a single node at a time. Block samplers are particularly beneficial when parameters in the model are highly correlated *a posteriori* (see, e.g., Roberts and Sahu 1997). The conditional distribution for a block $B \subseteq S$ of nodes, given the rest of nodes $V_{-B} = V \setminus B$, is

$$p(B \mid V_{-B}) \propto \prod_{b \in B} p(b \mid \mathrm{pa}(b)) \times \prod_{b \in B} \prod_{u \in \mathrm{ch}(b)} p(u \mid \mathrm{pa}(u)).$$

Block samplers can be parallelized in a straightforward manner: If we consider a block B as a single node, and define $\mathrm{ch}(B) = \cup_{b \in B} \mathrm{ch}(b)$, then the approach introduced above is immediately applicable, and we can exploit both opportunities for parallelization for block updates. A mixture of single node and block updaters can be used without complication.

In the seeds example it is possible to block together $\alpha_0, \alpha_1, \alpha_2, \alpha_{12}$. The block then has 21 children, and so our algorithm chooses to spread evaluation of their likelihood over multiple cores. The computation schedule remains identical to Table 1, but the block sampler waits until all the likelihoods corresponding to rows 7 to 10 of Table 1 are evaluated before determining each update for the $\{\alpha_0, \alpha_1, \alpha_2, \alpha_{12}\}$ block.

3. Implementation details

BUGS represents statistical models internally using a dynamic object-oriented data structure (Warford 2002) that is analogous to a DAG. The nodes of the graph are objects and the edges of the graph are pointers contained in these objects. Although the graph is specified in terms

of the parents of each node, BUGS identifies the children of each node and stores this as a list embedded in each parameter node. Each node object has a value and a method to calculate its probability density function. For observations and fixed hyperparameters the value is fixed and is read in from a data file; for parameters the value is variable and is sampled within a MCMC algorithm. Each MCMC sampling algorithm is represented by a class (Warford 2002) and a new sampling object of an appropriate class is created for each parameter in the statistical model. Each sampling object contains a link to the node (or block of nodes) in the graphical model that represents the parameter (or block of parameters) being sampled. One complete MCMC update of the model involves a traversal of a list of all these sampling objects, with each object's sampling method called in turn. Lunn *et al.* (2000) provide further background on the internal design of BUGS.

The **MultiBUGS** software consists of two distinct computer programs: a user interface and a computational engine. The computational engine is a small program assembled by linking together some modules of the **OpenBUGS** software plus a few additional modules to implement our parallelization algorithm. Copies of the computational engine run on multiple cores and communicate with each other using the message passing interface (MPI) protocol (Pacheco 1997), version 2.0. The user interface program is a slight modification (and extension) of the **OpenBUGS** software. The user interface program compiles an executable "worker program" that contains the computational engine required for a particular statistical model. It also writes out a file containing a representation of the data structures that specify the statistical model. It then starts a number of copies of the computational engine on separate computer cores. These worker programs then read in the model representation file to rebuild the graphical model and start generating MCMC samples using our distributed algorithms. The worker programs communicate with the user interface program via an MPI intercommunicator object. The user interface is responsible for calculating summary statistics of interest.

Both sources of parallelism described in Section 2.2 require only simple modifications of the data structures and algorithms used in the BUGS software. Each core keeps a copy of the current state of the MCMC, as well as two pseudo-random number generation (PRNG) streams (Wilkinson 2006): a "core-specific" stream, initialized with a different seed for each core; and "common" stream, initialized using the same seed on all cores. Initially, each core loads the sampling algorithm, the computation schedule, and the complete DAG, which is then altered as follows so that the overall computation yields the computation required for the original, complete DAG.

When the calculation of a parameter's likelihood is parallelized across cores, the list of children associated with a parameter on each core is thinned (pruned) so that it contains only the children in the corresponding partition component of $\mathrm{ch}(v)$. The BUGS MCMC sampling algorithm implementations then require only minor changes so that the partial likelihoods are communicated between cores. For example, a random walk Metropolis algorithm (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller 1953) is performed as follows: First, on each core, the prior factor and a partial likelihood contribution to the conditional distribution are calculated for the current value of the parameter. Each core then samples a candidate value. These candidates will be identical across cores, since the "common" PRNG stream is used. The prior and partial likelihood contributions are then calculated for the candidate value, and the difference between the two partial log-likelihood contributions can be combined across cores using the MPI function `Allreduce`. The usual Metropolis test can then be applied on each core in parallel using the "common" PRNG stream, after which the state of Markov

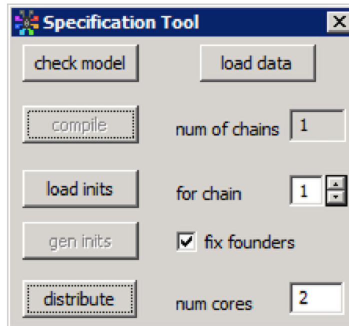

Figure 2: The specification tool in **MultiBUGS**, including the "distribute" button, which is used to initialize the parallelization.

chain is identical across cores. Computation of the prior factor and the Metropolis test is intentionally duplicated on every core because we found that the time taken to evaluate these quantities is usually shorter than the time taken to propagate their result across cores.

When a set of parameters W is sampled in parallel over the worker cores, the list of MCMC sampling objects is thinned on each core so that only parameters specified by the corresponding column of the computation schedule are updated on each core. The existing MCMC sampling algorithm implementations used in **OpenBUGS** can then be used without modification with each "core-specific" PRNG stream. The MPI function `Allgather` is used to send newly sampled parameters to each core. Note we need to run `Allgather` only after each core has sampled all of its assigned components in W, rather than after each component in W is sampled. For example, in the seeds example, we use `Allgather` after row 6. This considerably reduces message-passing overheads when the number of elements in W is large.

Running multiple chains is handled via standard MPI methods. If we have, say, two chains and eight cores, we partition the cores into two sets of four cores and set up separate MPI collective communicators (Pacheco 1997) for each set of cores for `Allreduce` and `Allgather` to use. Requests can be sent from the master to the workers using the intercommunicator and results returned. We find it useful to designate a special "lead worker" for each chain that we simulate. Each of these lead workers sends back sampled values to the master, where summary statistics can be collected. Only sampled values corresponding to quantities that the user is monitoring need to be returned to the master. This can considerably reduce the amount of communication between the workers and the master.

4. Basic usage of MultiBUGS

The procedure for running a model in **MultiBUGS** is largely the same as in **WinBUGS** or **OpenBUGS**. **MultiBUGS** adopts the standard BUGS language for specifying models, the core of which is common also to **WinBUGS**, **OpenBUGS**, **JAGS** and NIMBLE. A detailed tutorial on the use of BUGS can be found in, for example, Lunn, Jackson, Best, Thomas, and Spiegelhalter (2013).

An analysis is specified in **MultiBUGS** using the *Specification Tool* (*Model* › *Specification...*) by checking the syntax of a model (*check model*), loading the data (*load data*), compiling (*compile*) and setting up initial values (*load inits* and *gen inits*).

We then specify the total number of cores to distribute computation across by entering a number in the box labeled *num cores* (Figure 2) and then clicking *distribute*. This should be set at a value less than or equal to the number of processing cores available on your computer (the default is 2). If multiple chains are run, the cores will be divided equally across chains. We recommend that users experiment with different numbers of cores, since the setting that leads to fastest computation depends both on the specific model and data being analyzed and on the computing hardware being used. While increased parallelization will often result in faster computation, in some cases communication overheads will balloon to the point where parallelization gains are overturned. Furthermore, Amdahl's bound (Amdahl 1967) on the speed-up that is theoretically obtainable with increased parallelization may also be hit in some settings. Note that changing the number of cores will alter the exact samples obtained, since this affects the PRNG stream used to draw each sample (as described in Section 3).

Samples are drawn using the *Update Tool* (*Model* › *Update...*). The use of the *Sample Monitor Tool* (*Inference* › *Samples...*) to monitor parameters; to assess MCMC convergence, using, for example, the Brooks-Gelman-Rubin diagnostic (Gelman and Rubin 1992; Brooks and Gelman 1998); and to obtain results is the same as in **WinBUGS** and **OpenBUGS**. Analyses can be automated in **MultiBUGS** using the same simple procedural scripting language that is available in **OpenBUGS**. The new command `modelDistribute(C)` can be used to specify that parallelization should be across `C` cores; for details see *Manuals* › *MultiBUGS User Manual* › *Scripts and Batch-mode*.

4.1. Seeds example

The model, data and initial conditions for the seeds examples can be found within **MultiBUGS** in *Examples* › *Examples Vol I* › *Seeds: random effects logistic regression*. This is a simple model involving a small number of parameters and observations, so computation is already fast in **OpenBUGS** and is no faster in **MultiBUGS** (both take less than a second to do 1000 MCMC updates) because the benefit of parallelization is canceled out by communication overheads. However, for some more complicated models, **MultiBUGS** will be dramatically faster than **OpenBUGS**. We illustrate this with an example based on e-health data.

5. Illustration of usage with hierarchical e-health data

Our e-health example is based on a large linked database of methadone prescriptions given to opioid dependent patients in Scotland, which was used to examine the influence of patient characteristics on doses prescribed (Gao, Dimitropoulou, Robertson, McTaggart, Bennie, and Bird 2016; Dimitropoulou *et al.* 2017). This example is typical of many databases of linked health information drawn from primary care records, hospital records, prescription data and disease/death registries. Each data source often has a hierarchical structure, arising from regions, institutions and repeated measurements within individuals. Here, since we are unable to share the original dataset, we analyze a synthetic dataset, simulated to match the key traits of the original dataset.

The model includes 20,426 random effects in total, and was fitted to 425,112 observations. It is possible to fit this model using standard MCMC simulation in **OpenBUGS** but, unsurprisingly, the model runs extremely slowly and it takes 32 hours to perform a sufficient number of iterations (15,000) to satisfy standard convergence assessment diagnostics. In such data sets

it can be tempting to choose a much simpler and faster method of analysis, but this may not allow appropriately for the hierarchical structure or enable exploration of sources of variation. Instead it is preferable to fit the desired hierarchical model using MCMC simulation, while speeding up computation as much as possible by exploiting parallel processing.

The model code, data and initial conditions can be found within **MultiBUGS** in *Examples* › *Examples Vol IV* › *Methadone: an E-health model*.

5.1. E-health data

The data have a hierarchical structure, with multiple prescriptions nested within patients within regions. For some of the outcome measurements, person identifiers and person-level covariates are available (240,776 observations). These outcome measurements y_{ijk} represent the quantity of methadone prescribed on occasion k for person j in region i ($i = 1, \ldots, 8;\ j = 1, \ldots, J_i;\ k = 1, \ldots, K_{ij}$), and are recorded in the file `ehealth_data_id_available.txt`. Each of these measurements is associated with a binary covariate r_{ijk} (called `source.indexed`) that indicates the source of prescription on occasion k for person j in region i, with $r_{ijk} = 1$ indicating that the prescription was from a General Practitioner (family physician). No person identifiers or person-level covariates are available for the remaining outcome measurements (184,336 observations). We denote by z_{il} the outcome measurement for the lth prescription without person identifiers in region i ($i = 1, \ldots, 8;\ l = 1, \ldots, L_i$). These data are in the file `ehealth_data_id_missing.txt`. A binary covariate s_{il} (called `source.nonindexed`) indicates the source of the lth prescription without person identifiers in region i, with $s_{il} = 1$ indicating that the prescription was from a General Practitioner (family physician). The final data file, `ehealth_data_n.txt`, contains several totals used in the BUGS code.

5.2. E-health model

We model the effect of the covariates with a regression model, with regression parameter β_m corresponding to the mth covariate x_{mij} ($m = 1, \ldots, 4$), while allowing for within-region correlation via region-level random effects u_i, and within-person correlation via person-level random effects w_{ij}; source effects v_i are assumed random across regions.

$$y_{ijk} = \sum_{m=1}^{4} \beta_m x_{mij} + u_i + v_i r_{ijk} + w_{ij} + \varepsilon_{ijk}$$
$$u_i \sim \mathrm{N}(\mu_u, \sigma_u^2),\ v_i \sim \mathrm{N}(\mu_v, \sigma_v^2),\ w_{ij} \sim \mathrm{N}(\mu_w, \sigma_w^2),\ \varepsilon_{ijk} \sim \mathrm{N}(\mu_\varepsilon, \sigma_\varepsilon^2)$$

The means μ_w and μ_ε are both fixed to 0.

The outcome measurements z_{il} contribute only to estimation of regional effects u_i and source effects v_i.

$$z_{il} = \lambda + u_i + v_i s_{il} + \eta_{il}$$
$$\eta_{il} \sim \mathrm{N}(\mu_\eta, \sigma_\eta^2)$$

The error variance σ_η^2 represents a mixture of between-person and between-occasion variation. We fix the error mean $\mu_\eta = 0$. We assume uniform priors for $\sigma_u, \sigma_v, \sigma_w, \sigma_\varepsilon, \sigma_\eta$ on the range $\sigma_{\min} = 0$ to $\sigma_{\max} = 10$, and normal priors for $\beta_1, \ldots, \beta_4, \mu_u, \mu_v$ and λ with mean

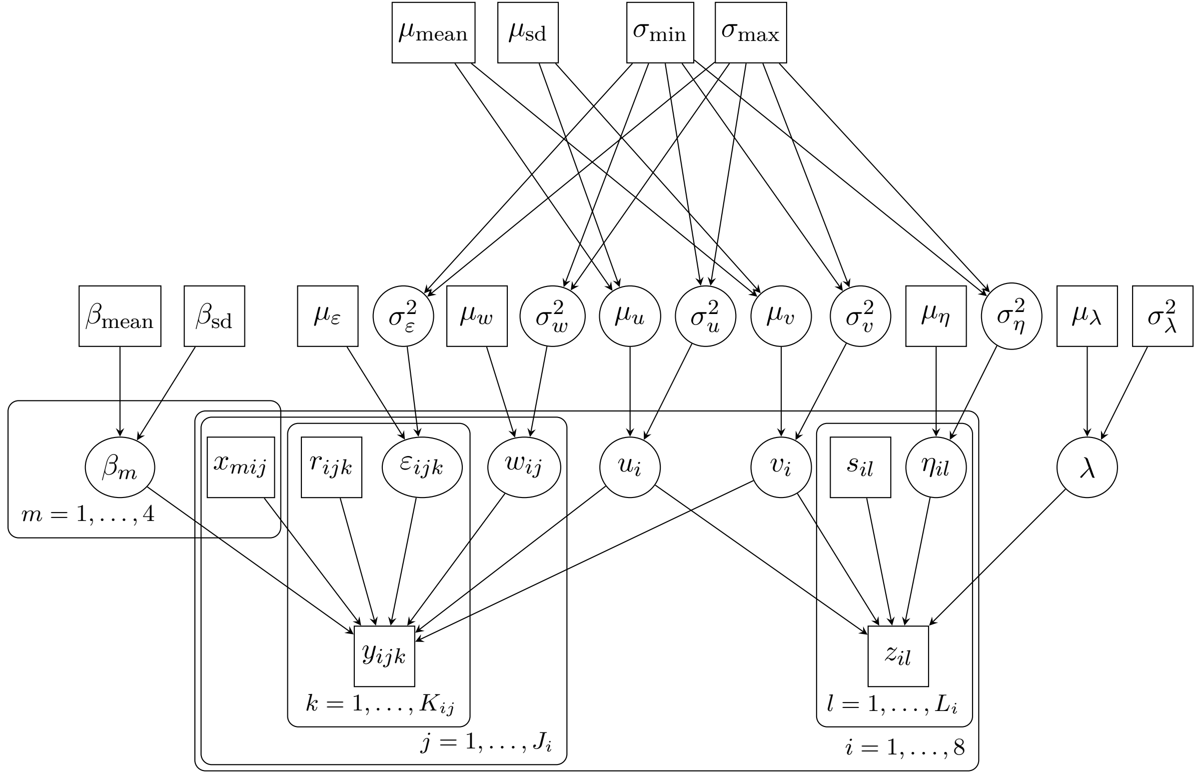

Figure 3: DAG representation of the e-health model.

$\beta_{\text{mean}} = \mu_{\text{mean}} = \mu_\lambda = 0$ and standard deviation $\beta_{\text{sd}} = \mu_{\text{sd}} = \mu_\lambda = 100$. Figure 3 is a DAG representation of this model.

The data have been suitably transformed so that fitting a linear model is appropriate. We do not consider alternative approaches to analyzing the data set. The key parameters of interest are the regression parameters $\beta_1, \ldots, \beta_4$ and the standard deviations σ_u and σ_v for the region and source random effects.

This model can be specified in BUGS as follows:

```
model {
  # Outcomes with person-level data available
  for (i in 1:n.indexed) {
    outcome.y[i] ~ dnorm(mu.indexed[i], tau.epsilon)
    mu.indexed[i] <- beta[1] * x1[i] +
                     beta[2] * x2[i] +
                     beta[3] * x3[i] +
                     beta[4] * x4[i] +
                     region.effect[region.indexed[i]] +
                     source.effect[region.indexed[i]] * source.indexed[i] +
                     person.effect[person.indexed[i]]
  }

  # Outcomes without person-level data available
```

```
  for (i in 1:n.nonindexed) {
    outcome.z[i] ~ dnorm(mu.nonindexed[i], tau.eta)
    mu.nonindexed[i] <- lambda +
                        region.effect[region.nonindexed[i]] +
                        source.effect[region.nonindexed[i]] *
                                      source.nonindexed[i]
  }

  # Hierarchical priors
  for (i in 1:n.persons) {
    person.effect[i] ~ dnorm(0, tau.person)
  }
  for (i in 1:n.regions) {
    region.effect[i] ~ dnorm(mu.region, tau.region)
    source.effect[i] ~ dnorm(mu.source, tau.source)
  }

  lambda ~ dnorm(0, 0.0001)
  mu.region ~ dnorm(0, 0.0001)
  mu.source ~ dnorm(0, 0.0001)

  # Priors for regression parameters
  for (m in 1:4) {
    beta[m] ~ dnorm(0, 0.0001)
  }

  # Priors for variance parameters
  tau.eta <- 1/pow(sd.eta, 2)
  sd.eta ~ dunif(0, 10)
  tau.epsilon <- 1/pow(sd.epsilon, 2)
  sd.epsilon ~ dunif(0, 10)
  tau.person <- 1/pow(sd.person, 2)
  sd.person ~ dunif(0, 10)
  tau.source <- 1/pow(sd.source, 2)
  sd.source ~ dunif(0, 10)
  tau.region <- 1/pow(sd.region, 2)
  sd.region ~ dunif(0, 10)
}
```

5.3. E-health initial values

For chain 1, we used the following initial values:

```
list(lambda = 0, beta = c(0, 0, 0, 0), mu.source = 0, sd.epsilon = 0.5,
  sd.person = 0.5, sd.source = 0.5, sd.region = 0.5, sd.eta = 0.5)
```

and for chain 2 we used:

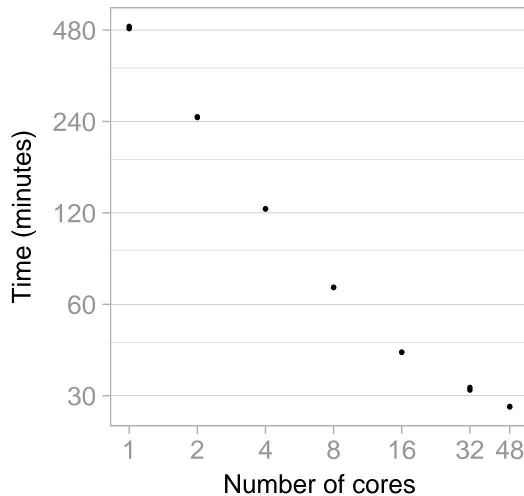

Figure 4: Run time against number of cores for 15,000 iterations of the e-health example model, running 2 chains simultaneously. The run time in each of 3 replicate runs are shown. Both time and number of cores are displayed on a log scale.

```
list(lambda = 0.5, beta = c(0.5, 0.5, 0.5, 0.5), mu.source = 0.5,
  sd.epsilon = 1, sd.person = 1, sd.source = 1, sd.region = 1, sd.eta = 1)
```

5.4. Parallelization in MultiBUGS

After setting the number of cores, the computation schedule chosen by **MultiBUGS** can be viewed in *Info* > *Show distribution*. **MultiBUGS** parallelizes sampling of all the person-level random effects w_{ij}, except for the component corresponding to the person with the most observations (176 observations); **MultiBUGS** parallelizes likelihood computation of this component instead. The likelihood computation of all the other parameters in the model is also parallelized, except for the mutually conditionally-independent sets $\{\mu_u, \mu_v\}$ and $\{\sigma_u^2, \sigma_v^2\}$, which are sampled in parallel in turn.

5.5. Run time comparisons across BUGS implementations

To demonstrate the speed-up possible in **MultiBUGS** using a range of number of cores, we ran two chains for 15,000 updates for the e-health example. This run length was chosen to mimic realistic statistical practice, since, after discarding the first 5,000 iterations as burn-in, visual inspection of chain-history plots and the Brooks-Gelman-Rubin diagnostic (Gelman and Rubin 1992; Brooks and Gelman 1998) indicated convergence. We ran the simulations (each replicated three times) on a 64-core machine consisting of four sixteen-core 2.4Ghz AMD Operon 6378 processors with 128GB shared RAM.

Figure 4 shows the run time against the number of cores on a log-log scale. Substantial time savings are achieved using **MultiBUGS**: On average using one core took 8 hours 10 minutes; using two cores took 4 hours and 8 minutes; and using 48 cores took only 28 minutes. In contrast, these simulations took 32 hours in standard single-core **OpenBUGS** 3.2.3; and 9 hours using **JAGS** 4.0.0 via R 3.3.1.

	mean	median	sd	MC_error	val2.5pc	val97.5pc	start	sample	ESS
beta[1]	-0.07124	-0.07137	0.01272	5.784E-4	-0.09561	-0.0461	5001	20000	483
beta[2]	-0.2562	-0.2563	0.02437	9.186E-4	-0.3036	-0.208	5001	20000	704
beta[3]	0.1308	0.1311	0.0114	5.7E-4	0.1085	0.1528	5001	20000	399
beta[4]	0.13	0.1305	0.0182	7.083E-4	0.09474	0.1651	5001	20000	660
sd.region	1.259	1.157	0.4606	0.005305	0.7024	2.445	5001	20000	7536
sd.source	0.3714	0.3417	0.1359	0.001611	0.2057	0.7153	5001	20000	7116

Table 2: Posterior summary table for the e-health example.

The scaling of performance with increasing number of cores is good up to sixteen cores and then displays diminishing gains. This may be due to inter core communication being much faster within each processor of 16 cores compared to across processors, or the diminishing returns anticipated by Amdahl's law (Amdahl 1967). Running only one chain approximately halved the run time for two chains.

5.6. Results

The posterior summary table we obtained is shown in Table 2.

6. Discussion

MultiBUGS makes Bayesian inference using multi-core processing accessible for the first time to applied statisticians working with the broad class of statistical models available in the BUGS language software. It adopts a pragmatic algorithm for parallelizing MCMC sampling, which we have demonstrated speeds up inference in a random-effects logistic regression model involving a large number of random effects and observations. While a large literature has developed proposing methods for parallelizing MCMC algorithms (see Section 1), a generic, easy-to-use implementation of these ideas has been heretofore lacking. Almost all users of BUGS language software will have a multi-core computer available, since desktop computers typically now have a moderate number (up to ten) of cores, and laptops typically have 2–4 cores. However, workstations with an even larger number of cores are now becoming available: For example, Intel's Xeon Phi x200 processor contains between 64 and 72 cores.

The magnitude of speed-up provided by **MultiBUGS** depends on the model and data being analyzed and the computer hardware being used. Two levels of parallelization can be used in **MultiBUGS**: Independent MCMC chains can be parallelized, and then computation within a single MCMC chain can be parallelized. The first level of parallelization will almost always be advantageous whenever sufficient cores are available, since no communication across cores is needed. The gain from second level of parallelization is problem specific: The gain will be largest for models involving parameters with a large number of likelihood terms and/or a large number of conditionally independent parameters. For example, **MultiBUGS** is able to parallelize inference for many standard regression-type models involving both fixed and random effects, especially with a large number of observations, since fixed effect regression parameters will have a large number of children (the observations), and random effects will typically be conditionally independent. For models without these features, the overheads

of the second level of parallelization may outweigh the gains on some computing hardware, meaning only the first level of parallelization is beneficial.

The mixing properties of the simulated MCMC chains are the same in **OpenBUGS** and **MultiBUGS**, because they use the same collection of underlying MCMC sampling algorithms. Models with severe MCMC mixing problems in **OpenBUGS** are thus not resolved in **MultiBUGS**. However, since **MultiBUGS** can speed-up MCMC simulation, it may be practicable to circumvent milder mixing issues by simply increasing the run length.

Several extensions and developments are planned for **MultiBUGS** in the future. First, at present **MultiBUGS** requires the Microsoft Windows operating system. However, most large computational clusters use the Linux operating system, so a version of **MultiBUGS** running on Linux is under preparation. Second, **MultiBUGS** currently loads data and builds its internal graph representation of a model on a single core. This process will need to be rethought for extremely large datasets and graphical models.

Acknowledgments

This work was supported by the UK Medical Research Council [program codes MC_UU_00002/2 (RJBG), MC_UU_12023/21 (RT), MC_UU_00002/11 (DDA and AT)]. We are grateful to Chris Jewell, Sylvia Richardson and Christopher Jackson for helpful discussions of this work; to the Associate Editor and Reviewers for their insightful comments; and also to all contributors to the BUGS project upon which **MultiBUGS** is based.

References

Amdahl GM (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, pp. 483–485. AFIPS Press, Reston.

Angelino E, Kohler E, Waterland A, Seltzer M, Adams RP (2014). "Accelerating MCMC via Parallel Predictive Prefetching." In *Proceedings of the Thirtieth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-14)*, pp. 22–31. Corvallis.

Bottolo L, Chadeau-Hyam M, Hastie DI, Zeller T, Liquet B, Newcombe P, Yengo L, Wild PS, Schillert A, Ziegler A, Nielsen SF, Butterworth AS, Ho WK, Castagné R, Munzel T, Tregouet D, Falchi M, Cambien F, Nordestgaard BG, Fumeron F, Tybjærg-Hansen A, Froguel P, Danesh J, Petretto E, Blankenberg S, Tiret L, Richardson S (2013). "**GUESS**-Ing Polygenic Associations with Multiple Phenotypes Using a GPU-Based Evolutionary Stochastic Search Algorithm." *PLOS Genetics*, **9**(8), e1003657. doi:10.1371/journal.pgen.1003657.

Bradford R, Thomas A (1996). "Markov Chain Monte Carlo Methods for Family Trees Using a Parallel Processor." *Statistics and Computing*, **6**(1), 67–75. doi:10.1007/bf00161575.

Breslow NE, Clayton DG (1993). "Approximate Inference in Generalized Linear Mixed Models." *Journal of the American Statistical Association*, **88**(421), 9–25. doi:10.1080/01621459.1993.10594284.

Brockwell AE (2006). "Parallel Markov Chain Monte Carlo Simulation by Pre-Fetching." *Journal of Computational and Graphical Statistics*, **15**(1), 246–261. doi:10.1198/106186006x100579.

Brooks SP, Gelman A (1998). "General Methods for Monitoring Convergence of Iterative Simulations." *Journal of Computational and Graphical Statistics*, **7**(4), 434–455. doi:10.1080/10618600.1998.10474787.

Calderhead B (2014). "A General Construction for Parallelizing Metropolis-Hastings Algorithms." *Proceedings of the National Academy of Sciences of the United States of America*, **111**(49), 17408–17413. doi:10.1073/pnas.1408184111.

Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "Stan: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. doi:10.18637/jss.v076.i01.

Crowder MJ (1978). "Beta-Binomial Anova for Proportions." *Journal of the Royal Statistical Society C*, **27**(1), 34–37. doi:10.2307/2346223.

de Valpine P, Turek D, Paciorek CJ, Anderson-Bergman C, Lang DT, Bodik R (2017). "Programming with Models: Writing Statistical Algorithms for General Model Structures with NIMBLE." *Journal of Computational and Graphical Statistics*, **26**(2), 403–413. doi:10.1080/10618600.2016.1172487.

Dimitropoulou P, Turner R, Kidd D, Nicholson E, Nangle C, McTaggart S, Bennie M, Bird S (2017). "Methadone Prescribing in Scotland: July 2009 to June 2013." Unpublished.

Gao L, Dimitropoulou P, Robertson JR, McTaggart S, Bennie M, Bird SM (2016). "Risk-Factors for Methadone-Specific Deaths in Scotland's Methadone-Prescription Clients between 2009 and 2013." *Drug and Alcohol Dependence*, **167**, 214–223. doi:10.1016/j.drugalcdep.2016.08.627.

Gelfand AE, Smith AFM (1990). "Sampling-Based Approaches to Calculating Marginal Densities." *Journal of the American Statistical Association*, **85**(410), 398–409. doi:10.1080/01621459.1990.10476213.

Gelman A, Rubin DB (1992). "Inference from Iterative Simulation Using Multiple Sequences." *Statistical Science*, **7**(4), 457–472. doi:10.1214/ss/1177011136.

Geman S, Geman D (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6**(6), 721–741. doi:10.1109/tpami.1984.4767596.

Goudie RJB, Presanis AM, Lunn D, De Angelis D, Wernisch L (2018). "Joining And Splitting Models with Markov Melding." *Bayesian Analysis*, **14**(1), 81–109. doi:10.1214/18-ba1104.

Jewell CP, Kypraios T, Neal P, Roberts GO (2009). "Bayesian Analysis for Emerging Infectious Diseases." *Bayesian Analysis*, **4**(3), 465–496. doi:10.1214/09-ba417.

Lauritzen SL, Dawid AP, Larsen BN, Leimer HG (1990). "Independence Properties of Directed Markov Fields." *Networks*, **20**(5), 491–505. doi:10.1002/net.3230200503.

Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter D (2013). *The BUGS Book: A Practical Introduction to Bayesian Analysis*. CRC Press, Boca Raton.

Lunn D, Spiegelhalter DJ, Thomas A, Best N (2009). "The BUGS Project: Evolution, Critique and Future Directions." *Statistics in Medicine*, **28**(25), 3049–3067. doi:10.1002/sim.3680.

Lunn D, Thomas A, Best N, Spiegelhalter DJ (2000). "**WinBUGS** – A Bayesian Modelling Framework: Concepts, Structure and Extensibility." *Statistics and Computing*, **10**(4), 325–337. doi:10.1023/a:1008929526011.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *The Journal of Chemical Physics*, **21**(6), 1087–1092. doi:10.1063/1.1699114.

Pacheco PS (1997). *Parallel Programming with MPI*. Morgan Kauffman, San Francisco.

Pearl J (2009). *Causality: Models, Reasoning, and Inference*. 2nd edition. Cambridge University Press, New York.

Plummer M (2017). *JAGS Version 4.2.0 User Manual*. URL http://mcmc-jags.sourceforge.net.

Roberts GO, Sahu SK (1997). "Updating Schemes, Correlation Structure, Blocking and Parameterization for the Gibbs Sampler." *Journal of the Royal Statistical Society B*, **59**(2), 291–317. doi:10.1111/1467-9868.00070.

Rosenthal JS (2000). "Parallel Computing and Monte Carlo Algorithms." *Far East Journal of Theoretical Statistics*, **4**, 207–236.

Scott SL, Blocker AW, Bonassi FV, Chipman HA, George EI, McCulloch RE (2016). "Bayes and Big Data: The Consensus Monte Carlo Algorithm." *International Journal of Management Science and Engineering Management*, **11**(2), 78–88. doi:10.1080/17509653.2016.1142191.

Spiegelhalter DJ, Thomas A, Best NG, Gilks W (1996). *BUGS 0.5: Bayesian Inference Using Gibbs Sampling – Manual (Version ii)*. Cambridge.

Thomas A (2006). "The BUGS Language." *R News*, **6**(1), 17–21. URL https://www.R-project.org/doc/Rnews/Rnews_2006-1.pdf.

Warford JS (2002). *Computing Fundamentals: The Theory and Practice of Software Design with BlackBox Component Builder*. Vieweg & Sohn, Braunschweig/Wiesbaden.

Whiley M, Wilson SP (2004). "Parallel Algorithms for Markov Chain Monte Carlo Methods in Latent Spatial Gaussian Models." *Statistics and Computing*, **14**(3), 171–179. doi:10.1023/b:stco.0000035299.51541.5e.

Wilkinson D (2006). "Parallel Bayesian Computation." In EJ Kontoghiorghes (ed.), *Handbook of Parallel Computing and Statistics*, pp. 477–508. Chapman & Hall/CRC, Boca Raton.

A. Technical algorithmic details

A.1. Identifying conditionally independent parameters

The following algorithm (called `find_conditionally_independent`) is used by **MultiBUGS** to identify sets of conditionally-independent parameters $W_1, \ldots, W_l \subseteq U$:

Input: $G = (E, V)$, a DAG; U, a set of nodes (with identical topological depth)
  $l \leftarrow 1$
  $M \leftarrow \varnothing$
  while $|U| > 0$ **do**
    for u in U **do**
      if $\text{ch}_G(u) \cap M = \varnothing$ **then**
        $W_l \leftarrow W_l \cup \{u\}$
        $U \leftarrow U \setminus \{u\}$
        $M \leftarrow M \cup \text{ch}_G(u)$
      end if
    end for
    $M \leftarrow \varnothing$
    $l \leftarrow l + 1$
  end while
Output: $\{W_1, \ldots, W_l\}$

A.2. Identifying parallelizable likelihoods

Nodes for which the likelihood calculations should be partitioned across cores are identified using the following algorithm, called `find_partial_product_parallel`:

Input: $G = (E, V)$, a DAG; C, a number of cores; h, a topological depth; $h^\star$, the maximum topological depth in G; T, a computation schedule; r, the current schedule row
  $U \leftarrow D_G^h$
  $\overline{\text{ch}} \leftarrow \text{mean}_{v \in S_G} |\,\text{ch}_G(v)|$
  for u in U **do**
    if $|\,\text{ch}_G(u)| > 2 \times \overline{\text{ch}}$ **or** $h^\star = 1$ **then**
      $r \leftarrow r + 1$
      for c in 1 to C **do**
        $T_{rc} \leftarrow u$
      end for
      $U \leftarrow U \setminus \{u\}$
    end if
  end for
Output: $\{T, U, r\}$

A.3. Building a computation schedule

The overall algorithm for allocating computation to cores is as follows:

Input: $G = (E, V)$, a DAG; C, a number of cores
  Initialize T, a table with C columns

$r \leftarrow 0$
$h^\star \leftarrow \max_{v \in S_G} d_G(v)$
for h in $h^\star$ to 1 **do**
  $\{T, U, r\} \leftarrow$ `find_partial_product_parallel`$(G, C, h, h^\star, T, r)$
  $\{W_1, \ldots, W_l\} \leftarrow$ `find_conditionally_independent`(G, U)
  for i in 1 to l **do**
    $c \leftarrow 0$
    for j in $\max_{w \in W_i} |\operatorname{ch}_G(w)|$ to 1 **do**
      for x in $\{w \in W_i : |\operatorname{ch}_G(w)| = j\}$ **do**
        if $c \bmod C = 0$ **then**
          $r \leftarrow r + 1$
          $c \leftarrow 0$
        end if
        $T_{r(c+1)} \leftarrow x$
        $c \leftarrow c + 1$
      end for
    end for
  end for
end for
Output: T

Affiliation:

Robert J. B. Goudie, Andrew Thomas
MRC Biostatistics Unit
School of Clinical Medicine
University of Cambridge
United Kingdom
E-mail: robert.goudie@mrc-bsu.cam.ac.uk, helsinkiant@gmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
October 2020, Volume 95, Issue 7 *Submitted:* 2017-09-28
doi:10.18637/jss.v095.i07 *Accepted:* 2018-12-03