Parallel Bayesian Additive Regression Trees

M. T. Pratola, H. Chipman, J. Gattiker, D. Higdon, R. McCulloch, and W. Rust

August 8, 2012

Abstract

Bayesian Additive Regression Trees (BART) is a Bayesian approach to flexible non-linear regression which has been shown to be competitive with the best modern predictive methods such as those based on bagging and boosting. BART offers some advantages. For example, the stochastic search Markov Chain Monte Carlo (MCMC) algorithm can provide a more complete search of the model space and variation across MCMC draws can capture the level of uncertainty in the usual Bayesian way. The BART prior is robust in that reasonable results are typically obtained with a default prior specification. However, the publicly available implementation of the BART algorithm in the R package BayesTree is not fast enough to be considered interactive with over a thousand observations, and is unlikely to even run with 50,000 to 100,000 observations. In this paper we show how the BART algorithm may be modified and then computed using single program, multiple data (SPMD) parallel computation implemented using the Message Passing Interface (MPI) library. The approach scales linearly in the number of processor cores, enabling the practitioner to perform statistical inference on massive datasets. Our approach can also handle datasets too massive to fit on any single data repository.

Contents

1 Introduction

The challenges confronting modern statistics are often very different from those faced by classical statistics. For instance, in today's applied problems one often has huge datasets involving millions of observations and hundreds or thousands of variables. Examples include the high-dimensional simulators found in computer experiments [ref], large complex genetic datasets in biostatistics [ref] and massive datasets of consumer behavior in computational advertising [ref]. In such problems, one challenge is modeling complex data structures in a manner that is both efficient and lends itself to useful inference.

The Bayesian Additive Regression Tree (BART) approach is presented in Chipman, George, and McCulloch (2010) (henceforth CGM). CGM consider the fundamental model

$$Y = f(x) + \epsilon, \qquad \epsilon \sim N(0, \sigma^2)$$

where $x = (x_1, \ldots, x_d)$ represents d predictors. The function f is represented as a sum of regression tree models $f(x) = \sum_{j=1}^{m} g_j(x)$ where $g_j(x)$ represents the contribution to the overall fit provided by the j^{th} regression tree. The number of regression tree models, m, is chosen to be large and the prior constrains the contribution of each regression tree model so that the overall fit is the sum of many small contributions. A Markov Chain Monte Carlo algorithm draws from the full joint posterior of all the regression tree models and σ.

CGM compare the out-of-sample predictive performance of BART with a boosting method, Random Forests, support vector machines, and Neural Nets and report that BART's performance is competitive. However, the data sets used for comparison have sample sizes of at most $n \approx 15,000$ (the drug-discovery example). All the BART computations reported in CGM are done using the function `bart` in the R package `BayesTree`. For the larger sample sizes often confronted in modern statistical applications, the `bart/BayesTree` implementation is hopelessly slow and uses large amounts of memory.

In this paper we describe an implementation of the BART method which is able to handle much larger data sets and is much faster for moderately sized data sets.

Our approach first simplifies the C++ representation of the regression tree models. We then simplify the Metropolis Hastings step in the BART MCMC algorithm presented in CGM. Finally, and crucially, we show that with our simplified BART, a Single Program, Multiple Data (SPMD) parallel implementation can be used to dramatically speed the computation. Our lean model representation is propagated across all processor cores while the data is split up into equal portions each of which is allocated to a particular core. Computations on the entire data set are done by computing results for each data portion on its own core, and then combining results across portions. The approach scales linearly in the number of processor cores, and can also handle datasets too massive to fit on any single data repository.

While prediction given a set of out-of-sample x's is fairly straightforward using BART, interpreting the fit may be difficult. Methods for interpretation often rely on designing a large number of x at which to make predictions. We also show how these prediction computations may be done with SPMD parallel computation.

The rest of the paper proceeds as follows. Section 2 reviews enough details of the MCMC algorithm for fitting BART so that the reader may understand how the three simplifications described above are carried out. Section 3 explains how we have implemented an efficient and parallel version of the BART algorithm. Section 4 compares the actual times needed to run the `BayesTree` BART algorithm, the BART implementation of this paper done serially, and the parallel BART implementation. Section 5 details the scalability of our parallel BART implementation. Section 6 discusses parallel computation for large numbers of predictions. Section 7 presents a few examples of analyzing data sets with large sample sizes using BART. Finally, we summarize our findings in Section 8

2 The BART Algorithm

We briefly review those aspects of the BART methodology in CGM necessary for the understanding of this paper. We borrow liberally from CGM.

2.1 The Sum of Trees Model

The BART model expresses the overall effect of the regressors as the sum of the contributions of a large number of regression trees. In this section we provide additional notation and detail so that we can describe our approach.

We parametrize a single regression tree model by the pair (T, M). T consists of the tree nodes and decision rules. The splits in T are binary so that each node is either terminal (at the bottom of the tree) or has a left and right child. Associated with each non-terminal node in T is a binary decision rule which determines whether an x descends the tree to the left or to the right. Typically, decision rules are of the form "go left if $x_j < c$" where x_j is the jth component of x. Let $b = |M|$ be the number of terminal nodes. Associated with the k^{th} terminal node is a number μ_k. M is the set of terminal node μ_k values: $M = (\mu_1, \ldots, \mu_b)$. To evaluate a single regression tree function, we drop an x down the tree T until it hits terminal node k. We then return the value μ_k. We use the function $g(x; T, M)$ to denote this returned value μ_k for input x and tree (T, M).

Let (T_j, M_j) denote the j^{th} regression tree model. Thus the $g_j(x)$ introduced in Section 1 is expressed as $g_j(x) \equiv g(x; T_j, M_j)$. Our sum of trees model is

$$Y = \sum_{j=1}^{m} g(x; T_j, M_j) + \epsilon, \qquad \epsilon \sim N(0, \sigma^2). \tag{1}$$

The prior specification for the parameter $((T_1, M_1), \ldots, (T_m, M_m), \sigma)$ is key to the BART methodology. Since our focus here is on computation, the reader is referred to CGM for the details.

2.2 The BART MCMC Algorithm

At the top level, the BART MCMC is a simple Gibbs sampler. Let $T_{(j)}$ denote all the trees except the j^{th} and define $M_{(j)}$ similarly. Our Gibbs sampler then consists of iterating the draws:

$$(T_j, M_j) \,|\, T_{(j)}, M_{(j)}, \sigma, y, \qquad j = 1, 2, \ldots, m \tag{2}$$

$$\sigma \,|\, T_1, \ldots T_m, M_1, \ldots, M_m, y.$$

The draw of σ is straightforward since given all the (T_j, M_j), (1) may be used to calculate ϵ, which can be treated as an observed quantity.

To make each of the m draws of (T_j, M_j), note that the conditional distribution $p(T_j, M_j | \, T_{(j)}, M_{(j)}, \sigma, y)$ depends on $(T_{(j)}, M_{(j)}, y)$ only through

$$R_j \equiv y - \sum_{k \neq j} g(x; T_k, M_k),$$

the $n-$vector of partial residuals based on a fit that excludes the jth tree. Conditionally, we have the single tree model

$$R_j = g(x; T_j, M_j) + \epsilon.$$

Thus, the m draws of (T_j, M_j) given $(T_{(j)}, M_{(j)}, \sigma, y)$ in (2) are equivalent to m draws from

$$(T_j, M_j) \,|\, R_j, \sigma, \qquad j = 1, \ldots, m,$$

and each one of these draws may be done using single tree methods.

Each single tree model draw of (T_j, M_j) is done using the approach of Chipman, George, and McCulloch (1998). The prior specification is chosen so that we can draw from the joint $(T_j, M_j) \,|\, R_j, \sigma$ by first analytically integrating out M_j and drawing from the marginal $T_j \,|\, R_j, \sigma$ and then drawing from the conditional $M_j \,|\, T_j, R_j, \sigma$.

The draws $T_j \,|\, R_j, \sigma$ are the heart of the algorithm. It is in these steps that the structure of the trees change. These draws are carried out using a Metropolis-Hastings step. Given a current tree structure, a modification is proposed and the modification is accepted or rejected (in which case the current structure is retained) according to the usual MH recipe.

In CGM, four different tree modification proposals are used. First, there are a complementary pair of BIRTH/DEATH proposals. A BIRTH proposal picks a terminal node and proposes a decision rule so that the node gives birth to two children. A DEATH step picks a pair of terminal nodes having the same parent node, and proposes eliminating them so that the parent becomes a terminal node.

The CHANGERULE move leaves the parent/child structure of the tree intact, but proposes a modification to the decision rule associated with one of the non-terminal nodes. The SWAP move picks a pair of non-terminal parent/child nodes and proposes swapping their decision rules.

The CGM method randomly picks one of the four tree modification proposals.

3 Efficient and Parallel Computation

In this section we detail how we have efficiently coded and simplified the BART algorithm and implemented single program, multiple data (SPMD) parallel computation.

Our first step was to code the tree models as simply as possible. Each node in a tree is represented as an instance of a C++ class. The class has only six data members: (i) a mean μ (ii) an integer v and integer c such the the decision rule is left if $x_v < c^{th}$ cutpoint (iii) a pointer to the parent node (iv) pointers to left and right children. Note that for a terminal node the pointers to left and right children are not assigned. For each component of x a discrete set of possible cutpoints are precalculated so that a cutpoint may be identified with an integer. This is the minimal information needed to represent a regression tree. A minimal representation speeds computation in that when trees are dynamically grown and shrunk as the MCMC runs, little computation is needed to make the modifications. In addition, in our SPMD implementation tree modifications must be propagated across the cores so that keeping the amount of information that must be sent small is important. A consequence of this lightweight representation is that some quantities that characterize a tree must be recomputed on demand, rather than stored. For example, to determine the depth of a node, pointers to parents, grandparents, etc must be followed. However these computations are fast, due in part to the typically small number of nodes in the trees used. Note that the implementation in the `R` function `bart` (package `BayesTree`), C++ classes are also used to represent nodes in a tree. However, the C++ classes in that implementation are much more complicated so that more computation and memory is needed to maintain them.

Our second simplification relative to CGM (and `bart/BayesTree`) is that only the BIRTH and DEATH proposals are used. In Chipman, George, and McCulloch (1998) it was found that using only BIRTH/DEATH moves resulted in an inferior MCMC exploration of the model space. Results obtained with different random number generator seeds could be dramatically different. However, these findings were in the context of a single regression tree model. BART behaves in a fundamentally different way, with individual trees that typically contain far fewer nodes. Small trees correspond to a more easily searched space. We have found, in many examples, that the fits obtained using only the BIRTH/DEATH moves are extremely similar to those obtained using all four moves discussed above in Section (2). It may be possible to efficiently code additional proposals, but our goal was to do things as simply as possible.

Finally, we outline our SPMD parallel computation. Given $p+1$ processor cores numbered $0, 1, 2, \ldots, p$, we split the data (y, x) into p (approximately) equally-sized portions, $\left(y_{(1)}, x_{(1)}\right), \ldots, \left(y_{(p)}, x_{(p)}\right)$ where the i^{th} data portion resides on core i. The current state of the regression tree models $((T_1, M_1), (T_2, M_2), \ldots, (T_m, M_m))$ is copied across all $p+1$ cores. The algorithm proceeds in a master-slave arrangement, where core 0 contains no observed data and only manages the MCMC sampler, while all computations involving the observed data take place on the p slave cores in parallel. Figure 1 illustrates the setup. Each large rectangle in the figure represents a core. Within each core, multiple trees are depicted representing the (T_j, M_j). However, core i only has data portion $y_{(i)}, x_{(i)}$.

As a simple example consider the draw $\sigma \,|\, T_1, \ldots T_m, M_1, \ldots, M_m, y$. To make this draw we just need the sufficient statistic $\sum_{i=1}^{n} \epsilon_i^2$, where n is the total number of observations and $\epsilon_i = y_i - \sum_{j=1}^{m} g(x_i; T_j, M_j)$. Since each core has copies of all the (T_j, M_j) it can compute the ϵ_i for its data portion and sum their squares. To make the draw of σ, the master node sends a request out to each slave core and each core responds with its portion of the total residual sum of squares. The master core adds up the residual sums of squares portion received from each slave and then draws σ. The ability to decompose sufficient statistics into sums of terms corresponding to different parts of the data enables this SPMD approach.

Consider the case of a BIRTH step. A particular terminal node of tree T_j in our sum of trees model has been chosen. A candidate decision rule (given by a choice of (v, c)) has been proposed. If we accept the move, the terminal node will be assigned the decision rule, and will be given a left and right child (and will hence cease to be a terminal node). To evaluate the MH acceptance probability of this proposed tree modification we need only know the sum of the partial residuals R_j for the observations assigned to the new left child and the sum for the observations in the new right child. This simplification is again the result of sufficiency under the assumption of normal errors. The master node manages the MH step. To compute the partial residual sums, the master sends out requests to the slaves and then sums the partial sums of the partial residuals. If the move is accepted, the master node then must propagate the change in T_j and M_j out to all the slaves.

The overall parallel MCMC sampler is summarized in Table 1. The calculations and communications required for each step of the MCMC are summarized by describing each operation performed on the Master node and on a given Slave node. The number of bytes for communication operations are specified as s(# bytes) and r(# bytes) for sends and receives respectively. Note that at each MCMC iteration, the BIRTH/DEATH proposals for $T_j \mid R_j, \sigma$ draws will involve at most m tree modifications that must be propagated across the slaves, depending on how many MH proposals are accepted. The sufficient statistics needed for the left/right nodes to undergo BIRTH/DEATH are denoted with subscripts l, r in the table. The $M_j \mid T_j, R_j, \sigma$ draws will involve propagating $\sum_{j=1}^{m} b_j$ new μ values across the slaves. The current value of σ need only be maintained on the master core, so the communication overheard in this step comes from receiving the partial residual sums-of-squares (RSS) from the slaves.

None of the parallel communications outlined depend on the sample size of the data set, and all but two are a small constant number of bytes. Conditional on the tree model being accessible on each core, all expensive computations involving the actual data (e.g. calculation of the partial sufficient statistics) are peformed independently on each slave core operating solely on the subset of data assigned to that core. Because of our lean model representation this algorithm is able to sample

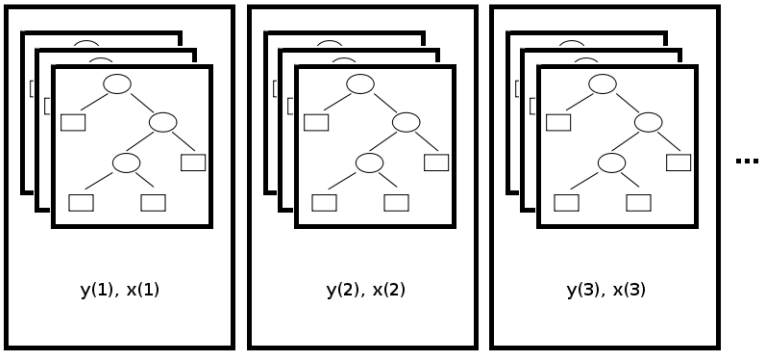

Figure 1: Each core has only a portion of the complete dataset (y, x) but all the model fit information (represented here by the trees).

from the posterior distribution $p((T_1, M_1), \ldots, (T_m, M_m), \sigma|y)$ efficiently with little communication overhead between computing cores.

The notions of sufficiency and the reduction of data to a statistical model figure prominently in this efficient implementation of BART. The large volumes of data are characterized by a few sufficient statistics and the simple statistical model, giving a compressed representation of the data that can be held in each cores local memory. This enables the quick exploration of the model space with the parallel BART algorithm.

4 Timing Results

Here we look at how this MCMC implementation speeds up with additional processors, considering a single dataset (x, y) where y is a 200,000-vector and x is 200,000$\times$40. These data are produced by a realization of the random function generator of Friedman (2001). The entries of x are iid draws from a $U[-1, 1]$ distribution. Briefly, given x_i, a row of x, y_i is an additive combination of randomly produced

Op (bytes)	**Master**	**Slave**	**Op** (bytes)
$T_j \vert R_j, \sigma \quad \forall j = 1, \ldots, m$			
BIRTH		BIRTH	
s(12)	Proposed split node, variable and cutpoint	Split node, variable and cutpoint	r(12)
		Calculate partial suff. stat.	
r(40)	Suff. stat.	$n_l, n_r, \sum R_l, \sum R_r, \sum R_l^2, \sum R_r^2$	s(40)
	MH Step		
s(28)	If accept BIRTH: node, variable, cutpoint, μ_l, μ_r	Update node, variable, cutpoint, μ_l, μ_r	r(28)
s(0)	Else reject BIRTH signal	Else reject BIRTH signal	r(0)
DEATH		DEATH	
s(8)	Nodes of children to kill	Nodes of children to kill	r(8)
		Calculate partial suff. stat.	
r(40)	Suff. stat.	$n_l, n_r, \sum R_l, \sum R_r, \sum R_l^2, \sum R_r^2$	s(40)
	MH Step		
s(28)	If accept DEATH: new terminal node and μ	Update new terminal node and μ	r(28)
s(0)	Else reject DEATH signal	Else reject DEATH signal	r(0)
$M_j \vert T_j, R_j, \sigma \quad \forall j = 1, \ldots, m$			
		Calculate partial suff. stat. for all b_j bottom nodes	
r($20b_j$)	Suff. stat.	$\{n_i, R_i, R_i^2\}_{i=1}^{b_j}$	s($20b_j$)
	Gibbs Step		
s($8b_j$)	Gibbs draw of M_j	Update M_j	r($8b_j$)
$\sigma \vert \cdot$			
		Calculate partial RSS	
r(8)	RSS	$\sum \epsilon^2$	s(8)
	Gibbs draw of σ		

Table 1: Summary of parallel MCMC sampler

normal kernels

$$y_i = \sum_{\ell=1}^{q} a_\ell q_\ell(x_i) + \epsilon_i. \tag{3}$$

The coefficients a_ℓ are iid $U[-1,1]$ draws. We take $q = 30$ and ϵ_i to be iid $N(0, \sigma^2)$ with $\sigma = 0.15$. The normal kernels $q_\ell(x)$ are determined by first randomly selecting a subset of components $[\ell]$ of x, giving $x_{[\ell]}$, randomly rotating these component directions with rotation matrix U_ℓ, and then stretching or dilating these rotated components according to the diagonal matrix D_ℓ

$$q_\ell(x) = \exp\left\{-\frac{1}{2}(x_{[\ell]} - \mu_\ell)^T U_\ell D_\ell^{-1} U_\ell^T (x_{[\ell]} - \mu_\ell)\right\}.$$

The mean vectors μ_ℓ are independent $U[-1,1]$ draws, same as the $x_{[\ell]}$'s. The diagonal matrix D_ℓ has diagonal entries d_k, with $\sqrt{d_k} \sim U[.1, 2]$.

This particular function realization produces components $[\ell]$ containing between 2 and 8 components of x in the $q = 30$ terms in (3). While the complexity of the function might have some effect on the computational time to carry out the MCMC, the timing is dominated by the size of the dataset (x, y).

Table 2: Time to complete 20K MCMC iterations for a 200,000×40 dataset. The number of processors includes the slave processor. The run time is wall clock time in seconds.

processors	run time (s)
2	347087
4	123802
8	37656
16	16502
24	9660
30	6303
40	4985
48	4477

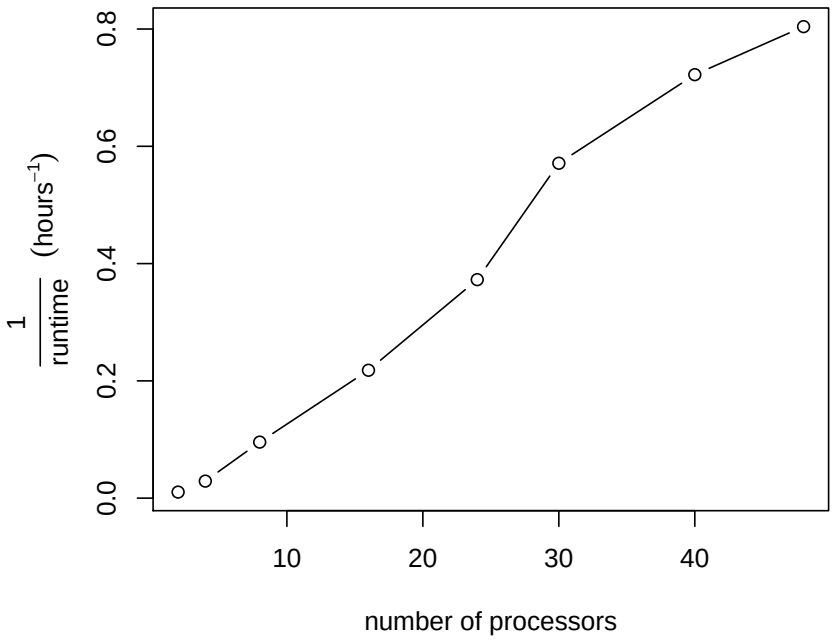

Figure 2: The inverse of wall clock time (in hours^{-1}) required to carry out 20K iterations of the parallel MCMC implementation for sampling the BART posterior distribution as a function of the number of processors. Here the dataset is 200000×40, produced by Friedman's random function generator.

Table 2 shows the time required to carry out the MCMC draws as a function of the number of processors, using this parallel implementation of BART. Here a total of 20,000 MCMC iterations were carried out for each timing run. As expected, the running time decreases with the number of processors, and the speed-up is nearly linear – the run time is about half when the number of processors is doubled. Figure 2 shows how the inverse of run time increases as a function of the number of processors.

5 Scalability

We analyze the scalability of the proposed MCMC algorithm building on the e-Isoefficiency approach presented in [ref Huerta et al 2007]. First, some basic quanti-

ties need to be defined. The speedup of an algorithm,

$$S(n,p) = \frac{T_{seq}}{T_{par}},$$

is the ratio of the times taken to run two instances of the algorithm. Typically, the speedup is regarded as the ratio of the sequential (or serial) algorithm's time to the parallel algorithm's time with p cores, as expressed above. Alternatively, the speedup could instead be measured relative to a smaller number of parallel cores. Here, n signifies the size of the problem, in our case the size of the dataset used in fitting the BART model. The efficiency,

$$E(n,p) = \frac{S(n,p)}{p},$$

is simply the speedup normalized to the number of cores used in the parallel version of the algorithm. For instance, if the algorithm were embarassingly parallel (i.e. no communication overhead) then the parallel time would be given by $\frac{T_{seq}}{p}$, resulting in an efficiency of 1.0.

The isoefficiency function is defined as

$$I(p,e) = n_e,$$

where e is the desired efficiency level with p cores and n_e represents the problem size to reach an efficiency e with p cores. This implicit function essentially relates the level of efficiency desired with the problem size, n_e, required to achieve that level of efficiency. Huerta et al. (2007) then define an algorithm to be e-Isoefficient as follows:

Definition: e-Isoefficiency (Huerta et al., 2007) *A problem is said to be e-Isoefficient in a system with p processors if* $\exists n_e$ *and* $p' > 0$ *such that for every* $p > p'$, $E(n_e, p) = e$.

In other words, if the efficiency of the algorithm with p processors can be maintained at the level e by increasing the problem size to some finite size n_e, then the algorithm is said to be e-Isoefficient, and thus scalable.

To determine if the BART MCMC algorithm is scalable in the sense of e-Isoefficiency, the speedup of the algorithm must be determined. Considering the complexity of an algorithm and assuming the time per operation is fixed, the runtime of an algorithm can be expressed as

$$\text{runtime} = \#\text{ of operations} \times \text{time per operation}$$

where the number of operations can be represented by the order of the algorithm and the time per operation can be thought of as a machine-specific constant that maps the algorithmic order to the algorithms runtime.

As a simple example, consider the draw of σ in our sampler which requires the residual sum of squares. In a serial algorithm, the order of this calculation is $O(n)$ and we can think of the runtime being $c_0 \times n$ for some constant c_0. In contrast, the parallel algorithm consists of the slave codes each calculating the partial residual sum of squares, a calculation of order $O\left(\frac{n}{p}\right)$ which happens simultaneously on all slaves. Subsequently, the results from the p slaves are collected and added on the master node, a calculation of order $O(p)$. The runtime of each of these components of the parallel algorithm can be thought of as $c_1 \times \frac{n}{p}$ and $c_2 \times p$. An approximation of the speedup can then be calculated as the fraction of these serial and parallel runtimes.

Applying these concepts to the entire MCMC sampler, it turns out that the speedup of BART is

$$S(n,p) \quad = \quad \frac{\alpha_1 mn + \alpha_2 mb + \alpha_3 n + \alpha_4 m}{\beta_1 m\frac{n}{p} + \beta_2 mb + \beta_3\frac{n}{p} + \beta 5m + \beta_6} \tag{4}$$

where $\frac{n}{p}$ is taken as the problem size on each slave node in the parallel implementation of the MCMC algorithm, b is a random variable representing the tree depth of all m trees under the prior, and the other variables are as defined previously. Here, the α's and β's are the unknown machine-specific parameters, and we have simplified the expression by collecting all like terms.

A unique characteristic of the proposed MCMC algorithm is that the speedup depends on the random variable b. Determining scalability in such a situation does not

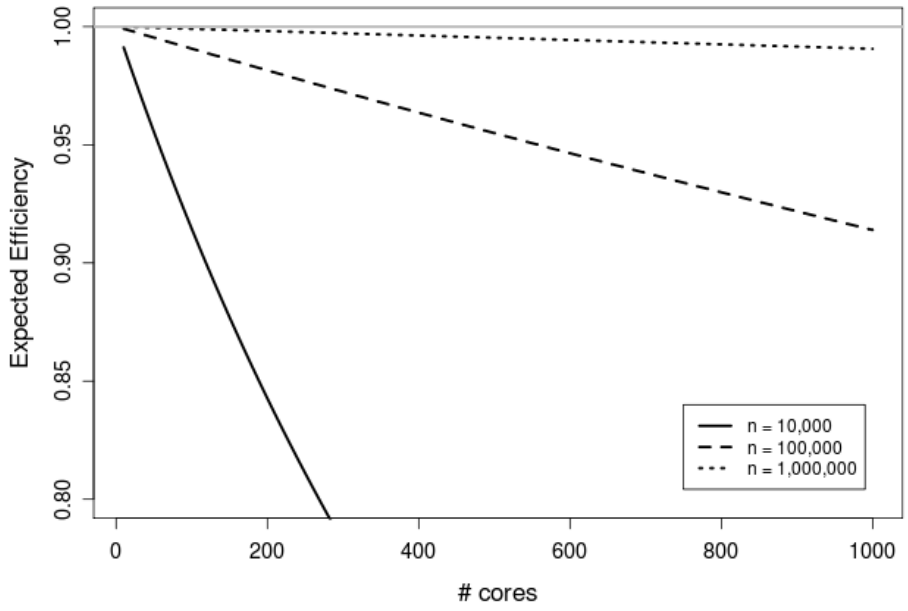

Figure 3: Expected efficiency of the BART MCMC algorithm for three problem sizes. The horizontal solid grey line represents the maximum efficiency of 1.0.

appear to have been explored in the literature. We consider the notion of Expected e-Isoefficiency, by determining the e-Isoefficiency when utilizing the expected speedup, where the expectation is taken with respect to the prior distribution on the number of terminal nodes, $\pi(b)$. Although this distribution is not known in closed form, samples from it can be easily constructed by drawing from the prior distribution of node depth.

To simplify the analysis, we take the machine-specific parameters as all equal to 1 which is akin to considering speedup in terms of algorithmic order rather than having a dependence on the actual machine(s) used. However, one could setup experiments to determine the values of these machine-specific parameters which may be useful in sizing a problem to the actual hardware available to the practitioner. The resulting expected efficiency curves for three problem sizes are shown in Figure 3. This plot indicates that the proposed algorithm is efficient and scalable, as the expected efficiency curves with an increased problem size always lie above efficiency curves with smaller problem sizes. That is, if we increase the number of cores, we can maintain a desired level of efficiency by increasing the problem size accordingly.

6 Prediction and Sensitivity Analysis

Predictions of the function f at unobserved input settings x^* can be constructed using the sampled posterior. For example, the posterior mean for f can be estimated by

$$\hat{f}(x^*) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{m} g_j(x^* | T_j^i, M_j^i)$$

where i indexes the N MCMC draws. Uncertainty bounds can be similarly calculated. Such computations are embarassingly parallel by simply subsetting the inputs $x^* = (x^*_{(1)}, \ldots, x^*_{(p)})$ over p computational cores and performing the predictions (or other calculations) for these subsets independently on each core.

Calculation of main effect functions (Cox, 1982; Sobol', 1993) or sensitivity indices Saltelli et al. (2008) can also be performed efficiently using the predicted response

with some minor communication overhead. Sobol's functional ANOVA decomposition uniquely represents $f(x)$ as the sum of increasingly complex terms

$$f(x) = f_0 + \sum_{k=1}^{d} f_k(x_k) + \sum_{1\le k<\ell\le d} f_{k\ell}(x_k, x_\ell) + \cdots + f_{1\cdots d}(x_1, \ldots, x_d).$$

The functions are computed by integrals over the x-space, so that

$$f_0 = \int_{[-1,1]^d} f(x)dx \text{ and } f_k(x_k) = \int_{[-1,1]^{d-1}} f(x)dx_{-k} - f_0,$$

where dx_{-k} includes all components of x but the k^{th}. The above integrals can easily be approximated via Monte Carlo integration, drawing x's uniformly over their domain, and using the posterior mean estimate $\hat{f}(x)$.

Similarly, the 1-way sensitivity index S_k for input k is

$$\begin{aligned} S_k = & \quad \frac{V_k}{V} \\ = & \quad \frac{Var_{x_k}\left(E_{x_{-k}}(f|x_k)\right)}{Var(f)} \\ \approx & \quad \frac{\int_{x_k} \hat{f}_k^2 dx_k}{\int_x \hat{f}^2(x)dx - \hat{f}_0^2}. \end{aligned}$$

Since the calculation of such indices involve integrals over the input space, there is some communication cost, but it is easily managed. Saltelli et al. (2008) approximate these calculations using Monte Carlo. For instance, the numerator can be calculated as

$$V_k \approx \sum_{j=1}^{n_s} \hat{f}(x_{j1}^a, x_{j2}^a, \ldots, x_{jd}^a) \times \hat{f}(x_{j1}^b, x_{j2}^b, \ldots, x_{jk}^a, \ldots, x_{jd}^b) - \hat{f}_0^2$$

using samples a, b each of size n_s from the input space. These samples hold a common, independent value for x_{jk}, but are otherwise independent. This calculation can be implemented in parallel by generating matrices $A_{(i)}$ and $B_{(i)}$ on the $i = 0, \ldots, p$ slave nodes where each row of a given matrix represents a randomly sampled point in the $d-$dimensional input space. Each matrix has approximately $\frac{n_s}{p}$ rows generated independently on each core. The samples can be drawn from a uniform distribution or a quasi Monte Carlo strategy may be used, such as a Sobol sequence. One must ensure that the matrices are unique on each core, so for a uniform sampling strategy

the random number generator seed must be different on all the cores. The integrals can then be approximated using the above summation by computing partial sums with the generated samples on each core and communicating these partial sums back to the master node. The master node then averages the partial results to arrive at the final Monte Carlo approximation of the sensitivity indices. This same parallel procedure can be used to approximate the total sensitivity index, S_k^T, of which the exact required calculations are described in detail in Saltelli et al. (2008).

7 Example

Taking the 200,000×40 dataset described in Section 4, we now show main effect functions from a sensitivity analysis and hold-out predictions produced by the MCMC carried out using this parallel implementation for BART. Scatterplots of y vs. each x_k are given in Figure 4 for the first 10,000 rows of (x, y).

The MCMC was carried out on 48 processors for 500,000 iterations, with the first 100,000 being discarded for burn-in. From these 400,000 post burn-in realizations of the posterior, an equally spaced sample of 600 BART surfaces, each consisting of a weighted sum of 200 trees, were saved to a file. These 400 posterior surfaces were then used to estimate main effect functions of the posterior mean surface and to predict a holdout set of $x's$.

The light lines in the Figure 4 show estimates of the mean shifted main effect functions $\hat{f}_0 + \hat{f}_k(x_k)$, $k = 1, \ldots, 40$, as described in Section 6. Finally, Figure 5 shows holdout predictions and accompanying two-standard deviation error bars for a randomly drawn collection of 10,000 x^*'s and corresponding y^*'s produced by the function machine realization described in Section 4. Unlike the 200,000×40 training sample (x, y), this hold out sample (x^*, y^*) does not have normal noise added to the function value to produce y^*. The RMSE for the holdout predictions is 0.082.

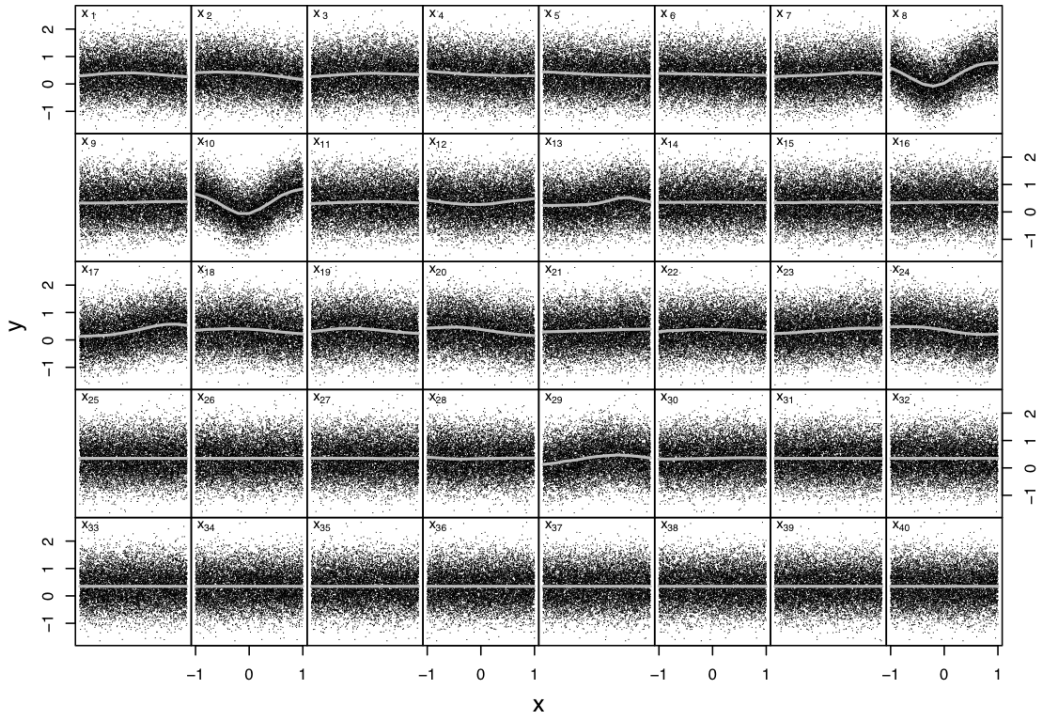

Figure 4: Scatterplots of each column of x and the function output y. Of the 200,000 data realizations in the dataset, only the first 10,000 are shown here. The light lines give the main effect functions estimated from the fitted BART surface.

8 Conclusion

We have presented and implemented a straight-forward SPMD approach for sampling the posterior distribution resulting from BART. In addition we have also constructed post processing parallel code to carry out basic sensitivity analyses and prediction. While timings and analyses were given for a rather small 200K dataset, this parallel implementation of BART has been successfully applied to millions of observations and has run on hundreds of cores.

The code, written in C++ using MPI, is available at [WEBSITE? ROB?]

References

Chipman, H., George, E., and McCulloch, R. (2010). "BART: Bayesian additive regression trees." *The Annals of Applied Statistics*, 4, 1, 266–298.

Cox, D. C. (1982). "An analytical method for uncertainty analysis of nonlinear output functions, with applications to fault-tree analysis." *IEEE Transactions in Reliability*, 31, 265–268.

Friedman, J. (2001). "Greedy function approximation: a gradient boosting machine." *Annals of Statistics*, 1189–1232.

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., and Tarantola, S. (2008). *Global sensitivity analysis: the primer*. Wiley Online Library.

Sobol', I. M. (1993). "Sensitivity analysis for non-linear mathematical models." *Mathematical Model. Comput. Exp.*, 1, 407–414.

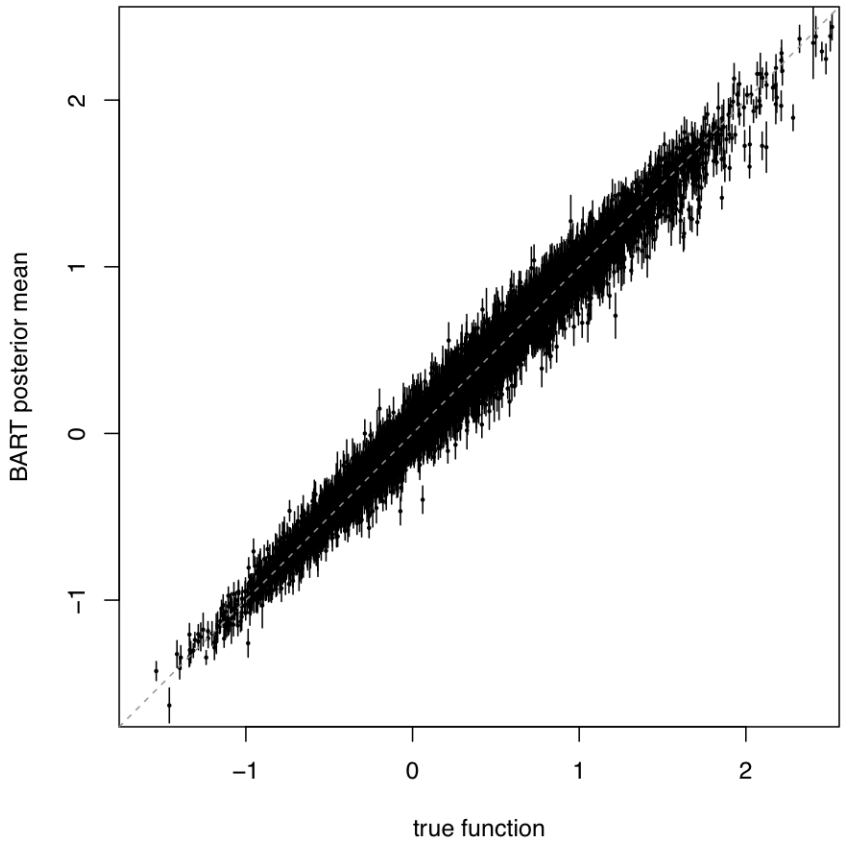

Figure 5: 10,000 holdout predictions and their corresponding two-standard deviation error bars. The holdouts, produced from the same realization of the random function machine described in Section 4, do not have additional white noise ϵ_i added to their values.