Efficient computing with BART

Rodney Sparapani
Medical College of Wisconsin

Robert McCulloch
Arizona State University

Abstract

This short article illustrates how to perform efficient computing with the **BART** package.

Keywords: Bayesian Additive Regression Trees, multi-threading, R, C++, forking, OpenMP.

1. Efficient computing with BART

If you had the task of creating an efficient implementation for a black-box model such as BART, which tools would you use? Surprisingly, linear algebra routines which are a traditional building block of scientific computing will be of little use for a tree-based method such as BART. So what is needed? Restricting ourselves to widely available off-the-shelf hardware and open-source sofware, we believe there are four key technologies necessary for a successful BART implementation.

- an object-oriented language to facilitate working with trees
- a parallel (or distributed) CPU computing framework for faster processing
- a high-quality parallel random number generator

In our implementation of BART, we pair the objected-oriented languages of R and C++ to satisfy these requirements. In this Section, we give a brief introduction to the concepts and technologies harnessed for efficient computing by our **BART** package.

1.1. A brief history of multi-threading

We now present a short synopsis of multi-threading. This is not meant to be exhaustive; rather, we only provide enough detail to explain the capability and popularity of multi-threading today. Multi-threading emerged rather early in the digital computer age with pioneers laying the research groundwork in the 1950s and 60s. In 1961, Burroughs released the B5000 which was the first commercial hardware capable of multi-threading (Lynch 1965). The B5000 performed asymmetric multiprocessing which is commonly employed in modern hardware via numerical co-processors and/or graphical processors today. In 1962, Burroughs released the D825 which was the first commercial hardware capable of symmetric multiprocessing (SMP) with CPUs (Anderson, Hoffman, Shifman, and Williams 1962). In 1967, Gene Amdahl derived the theoretical limits for multi-threading which came to be known as Amdahl's law (Amdahl 1967). If C is the number of CPUs and b is the fraction of work that can't be parallelized, then the gain due to multi-threading is $((1-b)/C+b)^{-1}$.

Let's fast-forward to the modern era of multi-threading. Although, not directly related to multi-threading, in 2000, Advanced Micro Devices (AMD) released the AMD64 specification that created a new 64-bit x86 instruction set which was capable of co-existing with 16-bit and 32-bit x86 legacy instructions. This was an important advance since 64-bit math is capable of addressing vastly more memory than 16-bit or 32-bit (2^{64} vs. 2^{16} or 2^{32}) and multi-threading inherently requires more memory resources. In 2003, version 2.6 of the Linux kernel incorporated full SMP support; prior Linux kernels had either no support or very limited/crippled support. From 2005 to 2011, AMD released a series of Opteron chips with multiple cores for multi-threading: 2 cores in 2005, 4 cores in 2007, 6 cores in 2009, 12 cores in 2010 and 16 cores in 2011. From 2008 to 2010, Intel entered the market with Xeon chips and their hyperthreading technology that allows each core to issue two instructions per clock cycle: 4 cores (8 threads) in 2008 and 8 cores (16 threads) in 2010. In this era including today, most off-the-shelf hardware available features 1 to 4 CPUs. Therefore, in the span of only a few years, multi-threading rapidly trickled down from servers at large firms to mass-market products such as desktops/laptops. For example, the consumer machine that **BART** is developed on, purchased in 2016, is capable of 8 threads (and hence many of the examples default to 8 threads).

1.2. Modern multi-threading software frameworks

In the late 1990s, the Message Passing Interface (MPI) (Walker and Dongarra 1996) was introduced which is the dominant distributed computing framework in use today (Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine, Castain, Daniel, Graham, and Woodall 2004), i.e., distributed meaning tasks which span multiple computers called nodes. R has some support for MPI provided in the **parallel** (Urbanek, Ripley, Tierney, and R Core 2017) package and additional support is provided by several other CRAN packages such as **snow** (Tierney, Rossini, Li, and Sevcikova 2016), **Rmpi** (Yu 2017) and **pbdMPI** (Chen, Ostrouchov, Schmidt, Patel, Yu, and R Core 2018). To support MPI, BART sofware was re-written with a simple, readable C++ object schema. Although, not an R package, this project was documented by Pratola, Chipman, Gattiker, Higdon, McCulloch, and Rust (2014). The current **BART** package source code is a descendent of the MPI BART project which is deprecated.

Furthermore, the current **BART** package no longer supports MPI; rather, the multi-threading available is now based on the OpenMP standard (Dagum and Menon 1998) and the **parallel** package. OpenMP takes advantage of modern hardware by performing multi-threading on single machines which often have multiple CPUs each with multiple cores. Currently, the **BART** package only uses OpenMP for parallelizing `predict` function calculations. The challenge with OpenMP, besides the programming per se, is that it is not widely available on all platforms. Operating system support can be detected by the GNU autotools (Calcote 2010) which define a C pre-processor macro if it is available, `_OPENMP`, or not. There are numerous exceptions for operating systems so it is difficult to generalize. But, generally, Microsoft Windows lacks OpenMP detection since the GNU autotools do not natively exist on this platform. And, Apple macOS lacks OpenMP support since the standard Xcode toolkit does not provide it. Thankfully, most Linux and UNIX distributions do provide OpenMP (although, macOS is technically a UNIX distribution, yet it is a notable exception in this regard). We provide the function `mc.cores.openmp` which returns > 0 (0) if the `predict` function is (not) capable of utilizing OpenMP.

The **parallel** package provides multi-threading via forking. Forking is available on Unix platforms, but not Windows (we use the term Unix to refer to UNIX, Linux and macOS since they are all in the UNIX family tree). The **BART** package uses forking for posterior sampling of the f function, and also for the `predict` function when OpenMP is not available. Except for `predict`, all functions that use forking start with `mc`. And, regardless of whether OpenMP or forking is employed, these functions except the argument `mc.cores` which controls the number of threads to be used. The **parallel** package provides the function `detectCores` which returns the number of threads that your hardware can support.

1.3. BART implementations on CRAN

Currently, there are four BART implementations on the Comprehensive R Archive Network (CRAN); see the Appendix for a tabulated comparative summary of their features.

BayesTree was the first released in 2006 (Chipman and McCulloch 2016). Reported bugs will be fixed, but no future improvements are planned; so, we suggest choosing one of the newer packages such as **BART**. The basic interface and workflow of **BayesTree** has strongly influenced the other packages which followed. However, the **BayesTree** source code is difficult to maintain and, therefore, improvements were limited leaving it with relatively fewer features than the other entries.

The next entrant is **bartMachine** which is written in java and was first released in 2013 (Kapelner and Bleich 2016). It provides advanced features like multi-threading, variable selection (Bleich, Kapelner, George, and Jensen 2014), a `predict` function, convergence diagnostics and missing data handling. However, the R to java interface can be challenging to deal with. R is written in C and Fortran, consequentally, functions written in java do not have a natural interface to R. This interface is provided by the **rJava** (Urbanek 2017) package which requires the Java Development Kit (JDK). Therefore, we recommend **bartMachine** only for those users who have a firm grounding in the java language and its tools in order to install/upgrade the package and get the best performance out of it.

The next entrant is **dbarts** which is written in C++ and was first released in 2014 (Dorie, Chipman, and McCulloch 2016). It is a clone of the **BayesTree** interface, but it does not share the source code; **dbarts** source has been re-written from scratch for efficiency and maintainability. **dbarts** is a drop-in replacement for **BayesTree**. However, **dbarts** has relatively fewer features than the other entries.

The **BART** package which is written in C++ was first released in 2017 (McCulloch, Sparapani, Gramacy, Spanbauer, and Pratola 2018). It provides advanced features like multi-threading, variable selection (Linero 2016), a `predict` function and convergence diagnostics. The source code is a descendent of the MPI BART project. Although, R is mainly written in C and Fortran (at the time of this writing, 39.2% and 26.8% lines of source code respectively), C++ is a natural choice for creating R functions since they are both object-oriented languages. The C++ interface to R has been seamlessly provided by the **Rcpp** package (Eddelbuettel, François, Allaire, Ushey, Kou, Russel, Chambers, and Bates 2011) which efficiently passes object references from R to C++ (and vice versa) as well as providing direct accesss to the R random number generator. The source code can also be called from C++ alone without an R instance where the random number generation is provided by either the standalone Rmath library (R Core 2017) or the C++ `random` Standard Template Library. Also, it is the only BART package to support categorical and time-to-event outcomes (Sparapani, Logan, McCulloch, and Laud 2016). It does not provide missing data imputation; rather, we recommend the **sbart** package for this niche which is performed by the so-called Sequential BART algorithm (Daniels and Singh 2017; Xu, Daniels, and Winterstein 2016) (**sbart** is also a descendent of MPI BART).

1.4. MCMC is embarrassingly parallel

In general, Bayesian Markov chain Monte Carlo (MCMC) posterior sampling is considered to be embarrassingly parallel (Rossini, Tierney, and Li 2007), i.e., since the chains only share the data and don't have to communicate with each other, parallel implementations are considered to be trivial. BART MCMC also falls into this class. Typical practice for Bayesian MCMC is to start in some initial state, perform a limited number of samples to generate a new random starting position and throw away the preceding samples which we call burn-in (the amount of burn-in in the **BART** package is controlled by the argument `nskip` which defaults to either 100 or 250). The total length of the chain returned is controlled by the argument `ndpost` which defaults to 1000. The theoretical gain due to multi-threading can be calculated by what we call the MCMC Corollary to Amdahl's Law. Let b be the burn-in fraction and C be the number of threads, then the gain limit is $((1-b)/C+b)^{-1}$. (As an aside, note that we can derive Amdahl's Law as follows where the amount of work done is in the numerator and elapsed time is in the denominator: $\frac{1-b+b}{(1-b)/C+b} = \frac{1}{(1-b)/C+b}$). For example, see the diagram in Figure 1 where the burn-in fraction, $b = \frac{100}{1100} = 0.09$, and the number of CPUs, $C = 5$, results in an elapsed time of only $((1-b)/C+b) = 0.27$ or a $((1-b)/C+b)^{-1} = 3.67$ fold reduction which is the gain in efficiency. In Figure 2, we plot theoretical gains on the y-axis and the number of CPUs on the x-axis for two settings: $b \in \{0.025, 0.1\}$.

1.5. Multi-threading and random access memory (RAM)

The IEEE standard 754-2008 (IEEE Computer Society 2008) specifies that every double-precision number consumes 8 bytes (64 bits). Therefore, it is quite simple to estimate the amount of random access memory (RAM) required to store a matrix. If A is $m \times n$, then the amount of RAM needed is $8 \times m \times n$ bytes. Large matrices held in RAM can present a challenge to system performance. If you consume all of the physical RAM, the system will "swap" segments out to virtual RAM which are disk files and this can degrade performance and possibly even crash the system. On Unix, you can monitor memory and swap usage with the `top` command-line utility. And, within R, you can determine the size of an object with the `object.size` function.

Mathematically, a matrix is represented as follows.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

R is a column-major language, i.e., matrices are laid out in consecutive memory locations by traversing the columns: $[a_{11}, a_{21}, \ldots, a_{12}, a_{22}, \ldots]$. R is written in C and Fortran where Fortran is a column-major language as well. However, C and C++ are row-major languages, i.e., matrices are laid out in consecutive memory locations by traversing the rows:

$[a_{11}, a_{12}, \ldots, a_{21}, a_{22}, \ldots]$. So, if you have written an R function in C/C++, then you need to be cognizant of the clash in paradigms (also note that R/Fortran array indexing goes from 1 to m while C/C++ indexing goes from 0 to $m-1$). As you might surmise, this is easily addressed with a transpose, i.e., instead of passing A from R to C/C++, pass A^t.

R is very efficient in passing objects; rather, than passing an object (along with all of its memory consumption) on the stack, it passes objects merely by a pointer referencing the original memory location. However, R follows copy-on-write memory allocation, i.e., all objects present in the parent thread can be read by a child thread without a copy, but when an object is altered/written by the child, then a new copy is created in memory. Therefore, if we pass A from R to C/C++, and then transpose, we will create multiple copies of A consuming $8 \times m \times n \times C$ where C is the number of children. If A is a large matrix, then you may stress the system's limits. The simple solution is for the parent to create the transpose before passing A and avoiding the multiple copies, i.e., `A <- t(A)`. And this is the philosophy that the **BART** package follows.

1.6. Multi-threading: interactive and batch processing

Interactive jobs must take precedence over batch jobs to prevent the user experience from suffering high latency. For example, have you ever experienced a system slowdown while you are typing and the display of your keystrokes can not keep up; this should never happen and is the sign of something amiss. With large multi-threaded jobs, it is surprisingly easy to naively degrade system performance. But, this can easily be avoided by operating system support provided by R. In the **tools** package (Hornik and Leisch 2017), there is the `psnice` function. Paraphrased from the `?psnice` help page.

> Unix has a concept of process priority. Priority is assigned values from 0 to 39 with 20 being the normal priority and (counter-intuitively) larger numeric values denoting lower priority. Adding to the complexity, there is a "nice" value, the amount by which the priority exceeds 20. Processes with higher nice values will receive less CPU time than those with normal priority. Generally, processes with nice 19 are only run when the system would otherwise be idle.

Therefore, by default, the **BART** package children have their nice value set to 19.

1.7. Continuous BART: serial and parallel implementations

Here we present snippets of R code to run BART in serial and parallel for a continuous outcome which we call continuous BART. While we only demonstrate continuous outcomes, the other outcomes are as similarly handled by the **BART** package as possible to present a consistent interface. The serial function is `wbart` and the parallel, `mc.wbart`. The 'w' in the name stands for weighted since you can provide known weights (with the `w` argument) for the following model: $y_i \sim \mathrm{N}\left(f(\boldsymbol{x}_i,\ w_i^2\sigma^2\right)$ where $(f, \sigma) \overset{\text{prior}}{\sim}$ BART and $i = 1, \ldots, N$ indexes subjects. Now, we can perform the calculations in serial,

```
set.seed(99); post <- wbart(x.train, y.train, ..., ndpost=M)
```

or in parallel (when said support is available),

```
post <- mc.wbart(x.train, y.train, ..., ndpost=M, mc.cores=8, seed=99).
```

Notice the difference in how the seed is set; we will return to this detail later on. The **BART** package allows `x.train` (and `x.test`) to be provided as matrices or data frames, but for simplicity we present them as matrices.

$$\text{Input: } \texttt{x.train} \text{ and, optionally, } \texttt{x.test}\text{: } \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_N \end{bmatrix} \text{ where } \boldsymbol{x}_i \text{ is the } i^{th} \text{ row}$$

$$\text{Output: } \texttt{post\$yhat.train} \text{ and } \texttt{post\$yhat.test}\text{: } \begin{bmatrix} \hat{y}_{11} & \dots & \hat{y}_{N1} \\ \vdots & \vdots & \vdots \\ \hat{y}_{1M} & \dots & \hat{y}_{NM} \end{bmatrix} \text{ where } \hat{y}_{im} = f_m(\boldsymbol{x}_i)$$

The `post` object returned is of type `wbart` which is essentially a list. There are other items returned in the list, but here we only focus on `post$yhat.train` and `post$yhat.test`; the latter only being returned if `x.test` is provided. In the above display, $m = 1, \dots, M$ are the MCMC samples which are the rows of `post$yhat.train` and `post$yhat.test`. Note that each outcome has a different return type, i.e., `post` object of type `wbart` (continuous), `pbart` (binary probit), `lbart` (binary logistic), `survbart` (survival analysis), `criskbart` (competing risks) or `recurbart` (recurrent events).

1.8. Continuous BART: predicting with a previous fit

Often when we are fitting a BART model, we have not specified an `x.test` matrix of hypothetical values for the evaluation of f. For fire-and-forget packages like **BayesTree** and **dbarts**, we would have to re-fit the model every time we want to evaluate **x.test** which can be very time-consuming. Therefore, the **BART** package takes a unique approach: it returns the ensemble of trees in the `post` object for later use; specifically, they are encoded in an ASCII character string, `post$treedraws$trees`. This allows us to construct `x.test` after the fact which is often convenient when it is large since we can partition it into smaller chunks. Then we can evaluate predictions via the S3 method `predict.wbart`. The predictions are generated in serial by default,
```
pred <- predict(post, x.test, ...)
```
but can be parallelized (when said support is available),
```
pred <- predict(post, x.test, mc.cores=8, ...).
```

$$\text{Input: } \texttt{x.test}\text{: } \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \end{bmatrix} \text{ where } \boldsymbol{x}_i \text{ is the } i^{th} \text{ row and } i = 1, \dots, Q$$

$$\text{Output matrix: } \begin{bmatrix} \hat{y}_{11} & \dots & \hat{y}_{Q1} \\ \vdots & \vdots & \vdots \\ \hat{y}_{1M} & \dots & \hat{y}_{QM} \end{bmatrix} \text{ where } \hat{y}_{im} = f_m(\boldsymbol{x}_i)$$

In the above display, $m = 1, \ldots, M$ are the MCMC samples which are the rows of the output matrix.

1.9. Binary Trees

BART relies on an ensemble of H binary trees. We exploit the tree metaphor to its fullest. Each of these trees grows from the ground up starting out as a root node. The root node is generally a branch decision rule, but it doesn't have to be; occasionally there is only a root terminal node and its corresponding leaf output value. If the root is a branch decision rule, then it spawns a left and a right node which each can be either a branch decision rule or a terminal leaf value and so on. In tree, T, there are C nodes which are made of B branches and L leaves: $C = B + L$. There is an algebraic relationship between the number of branches and leaves which we express as $C = 2L - 1$.

The ensemble of trees is encoded in an ASCII string which is returned in the `treedraws$trees` list item. This string can be easily exported and imported by R with the following:

```
write(post$treedraws$trees, 'trees.txt')
tc <- textConnection(post$treedraws$tree)
trees <- read.table(file=tc, fill=TRUE, row.names=NULL, header=FALSE,
col.names=c('node', 'var', 'cut', 'leaf'))
close(tc)
```

The string is encoded via the following binary tree notation. The first line is an exception which has the number of MCMC samples, M, in the field `node`; the number of trees, H, in the field `var`; and the number of variables, P, in the field `cut`. For the rest of the file, the field `node` is used for the number of nodes in the tree when all other fields are `NA`; or for a specific node when the other fields are present. The nodes are numbered in relation to the tier level, `t=floor(log2(node))`, as follows.

Tier t	$2^t + 0$			$\ldots$		$2^t + 2^t - 1$	
$\vdots$							
2	4		5		6		7
1		2				3	
0				1			

The `var` field is the variable in the branch decision rule which is encoded $0, \ldots, P - 1$ as a C/C++ array index (rather than an R index). Similarly, the `cut` field is the cutpoint of the variable in the branch decision rule which is encoded $0, \ldots, c_j - 1$ for variable j; note that the cutpoints are returned in the `treedraws$cutpoints` list item. The terminal leaf output value is contained in the field `leaf`. It is not immediately obvious which nodes are branches vs. leaves since it appears that the `leaf` field is given for both branches and leaves. Leaves are always associated with `var=0` and `cut=0`; however, note that this is also a valid branch variable/cutpoint since these are C/C++ indices. The key to discriminating between branches and leaves is the algebraic relationship between a branch, v, at tier t leading to its left, l, and right, r, nodes at tier $t + 1$. If $v = 2^t + k$, then $l = 2^{t+1} + 2k$ and $r = 2^{t+1} + 2k + 1$, i.e., for each node, besides root, you can determine from which branch it arose and those nodes that are not a branch (since they have no leaves) are necessarily leaves.

1.10. Creating a BART executable

Occasionally, you may need to create a BART executable that you can run without an R instance. This is especially useful if you need to include BART in another C++ program. Or, when you need to debug the **BART** package C++ source code which is more difficult to do when you are calling the function from R. Several examples of these are provided with the **BART** package. With R, you can find the `Makefile` and the weighted BART example with `system.file('cxx-ex/Makefile', package='BART')` and `system.file('cxx-ex/wmain.cpp', package='BART')` respectively. Note that these examples require the installation of the standalone Rmath library (R Core 2017) which is contained in the R source code distribution. Rmath provides common R functions and random number generation, e.g., `pnorm` and `rnorm`. You will likely need to copy the `cxx-ex` directory to your workspace. Once done, you can build and run the weighted BART executable example from the command line as follows.

```
> make wmain.out ## to build
> ./wmain.out ## to run
```

By default, these examples are based on the Rmath random number generator. However, you can specify the C++ Standard Template Library random number generator (contained in the STL `random` header file) by uncommenting the following line in the `Makefile` (by removing the pound, #, symbols):

```
## CPPFLAGS = -I. -I/usr/local/include -DMATHLIB_STANDALONE -DRNG_random
```

(which still requires Rmath for other purposes). These examples were developed on Linux and macOS, but they should be readily adaptable to UNIX and Windows as well.

References

Amdahl G (1967). “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.” In *AFIPS Conference Proceedings*, volume 30, pp. 483–5.

Anderson JP, Hoffman SA, Shifman J, Williams RJ (1962). “D825 - a multiple-computer system for command and control.” In *AFIPS Conference Proceedings*, volume 24.

Bleich J, Kapelner A, George EI, Jensen ST (2014). “Variable selection for BART: an application to gene regulation.” *The Annals of Applied Statistics*, **8**(3), 1750–1781.

Calcote J (2010). *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press.

Chen W, Ostrouchov G, Schmidt D, Patel P, Yu H, R Core (2018). *pbdMPI: Programming with Big Data - Interface to MPI*. [`https://CRAN.R-project.org/package=pbdMPI`].

Chipman H, McCulloch R (2016). *BayesTree: Bayesian Additive Regression Trees*. [`https://CRAN.R-project.org/package=BayesTree`].

Dagum L, Menon R (1998). “OpenMP: an industry standard API for shared-memory programming.” *IEEE computational science and engineering*, **5**(1), 46–55.

Daniels M, Singh A (2017). *sbart: Sequential BART for Imputation of Missing Covariates.* [https://cran.r-project.org/package=sbart].

Dorie V, Chipman H, McCulloch R (2016). *dbarts: Discrete Bayesian Additive Regression Trees Sampler.* [https://CRAN.R-project.org/package=dbarts].

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2011). "Rcpp: Seamless R and C++ integration." *Journal of Statistical Software*, **40**(8), 1–18.

Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain R, Daniel D, Graham R, Woodall T (2004). "Open MPI: Goals, concept, and design of a next generation MPI implementation." In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 97–104. Springer.

Hornik K, Leisch F (2017). *tools: Tools for Package Development.* [https://CRAN.R-project.org].

IEEE Computer Society (2008). IEEE Std 754-2008.

Kapelner A, Bleich J (2016). *bartMachine: Bayesian Additive Regression Trees.* [https://CRAN.R-project.org/package=bartMachine].

Linero AR (2016). "Bayesian regression trees for high dimensional prediction and variable selection." *Journal of the American Statistical Association*, (<doi:10.1080/01621459.2016.1264957>).

Lynch J (1965). "The Burroughs B8500." *Datamation*, pp. 49–50.

McCulloch R, Sparapani R, Gramacy R, Spanbauer C, Pratola M (2018). *BART: Bayesian Additive Regression Trees.* [https://CRAN.R-project.org/package=BART].

Pratola MT, Chipman HA, Gattiker JR, Higdon DM, McCulloch R, Rust WN (2014). "Parallel Bayesian additive regression trees." *Journal of Computational and Graphical Statistics*, **23**(3), 830–52.

R Core (2017). *Mathlib: A C Library of Special Functions.* [https://cran.r-project.org/doc/manuals/r-release/R-admin.html#The-standalone-Rmath-library].

Rossini AJ, Tierney L, Li N (2007). "Simple parallel statistical computing in R." *Journal of computational and Graphical Statistics*, **16**(2), 399–420.

Sparapani RA, Logan BR, McCulloch RE, Laud PW (2016). "Nonparametric survival analysis using Bayesian Additive Regression Trees (BART)." *Statistics in medicine*, **35**(16), 2741–53.

Tierney L, Rossini A, Li N, Sevcikova H (2016). *snow: Simple Network of Workstations.* [https://CRAN.R-project.org/package=snow].

Urbanek S (2017). *rJava: Low-Level R to Java Interface.* [https://CRAN.R-project.org/package=rJava].

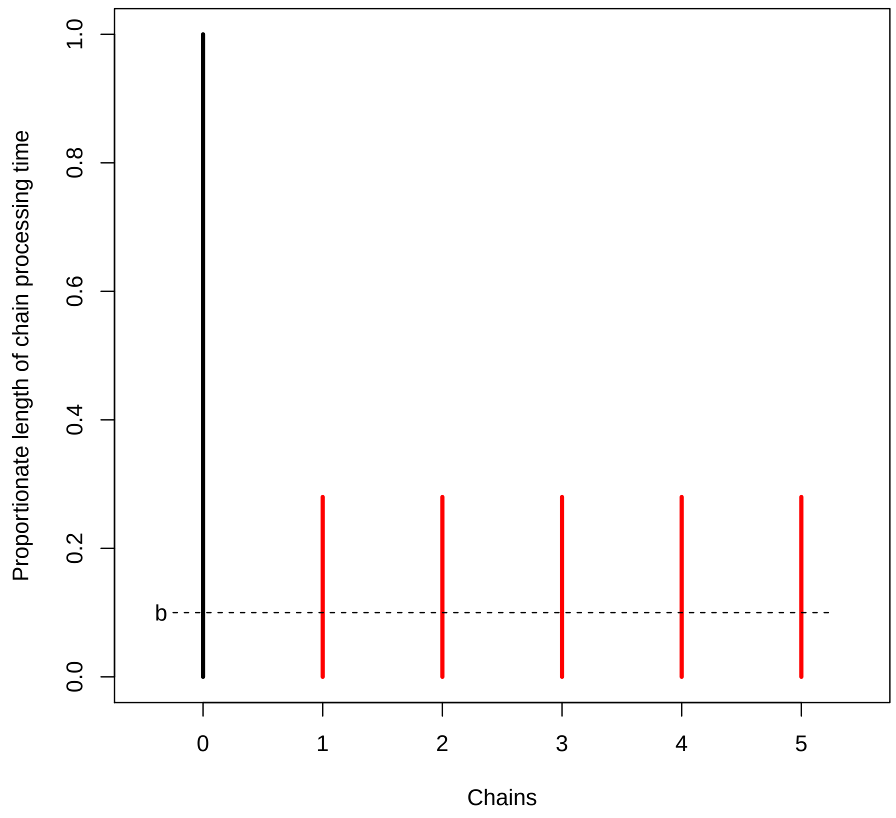

Figure 1: The theoretical gain due to multi-threading can be calculated by Amdahl's Law. Let b be the burn-in fraction and C be the number of threads, then the gain limit is $((1-b)/C+b)^{-1}$. In this diagram, the burn-in fraction, $b = \frac{100}{1100} = 0.09$, and the number of CPUs, $C = 5$, results in an elapsed time of only $((1-b)/C+b) = 0.27$ or a $((1-b)/C+b)^{-1} = 3.67$ fold reduction which is the gain in efficiency.

Urbanek S, Ripley B, Tierney L, R Core (2017). *parallel: Support for Parallel Computation.* [https://CRAN.R-project.org].

Walker DW, Dongarra JJ (1996). "MPI: a standard message passing interface." *Supercomputer*, **12**, 56–68.

Xu D, Daniels MJ, Winterstein AG (2016). "Sequential BART for imputation of missing covariates." *Biostatistics*, **17**(3), 589–602.

Yu H (2017). *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface).* [https://CRAN.R-project.org/package=Rmpi].

Category	**BayesTree**	**bartMachine**	**dbarts**	**BART**
First release	2006	2013	2014	2017
Authors	Chipman & McCulloch	Kapelner & Bleich	Dorie, Chipman & McCulloch	McCulloch, Sparapani Gramacy, Spanbauer & Pratola
Source code	C++	java	C++	C++
CRAN dependencies	nnet	rJava, car, randomForest, missForest		Rcpp
Tree transition proposals	4	3	4	3
Multi-threaded	No	Yes	No	Yes
`predict` function	No	Yes	No	Yes
Variable selection	No	Yes	No	Yes
Continuous outcomes	Yes	Yes	Yes	Yes
Dichotomous outcomes with Normal latents	Yes	Yes	Yes	Yes
Dichotomous outcomes with Logistic latents	No	No	No	Yes
Categorical outcomes	No	No	No	Yes
Time-to-event outcomes	No	No	No	Yes
Convergence diagnostics	No	Yes	No	Yes
Thinning	Yes	No	Yes	Yes
Cross-validation	No	Yes	Yes	No
Missing data handling	No	Yes	No	No
Partial dependence plots	Yes	Yes	Yes	No
Citations	Chipman and McCulloch (2016)	Kapelner and Bleich (2016)	Dorie *et al.* (2016)	McCulloch *et al.* (2018)

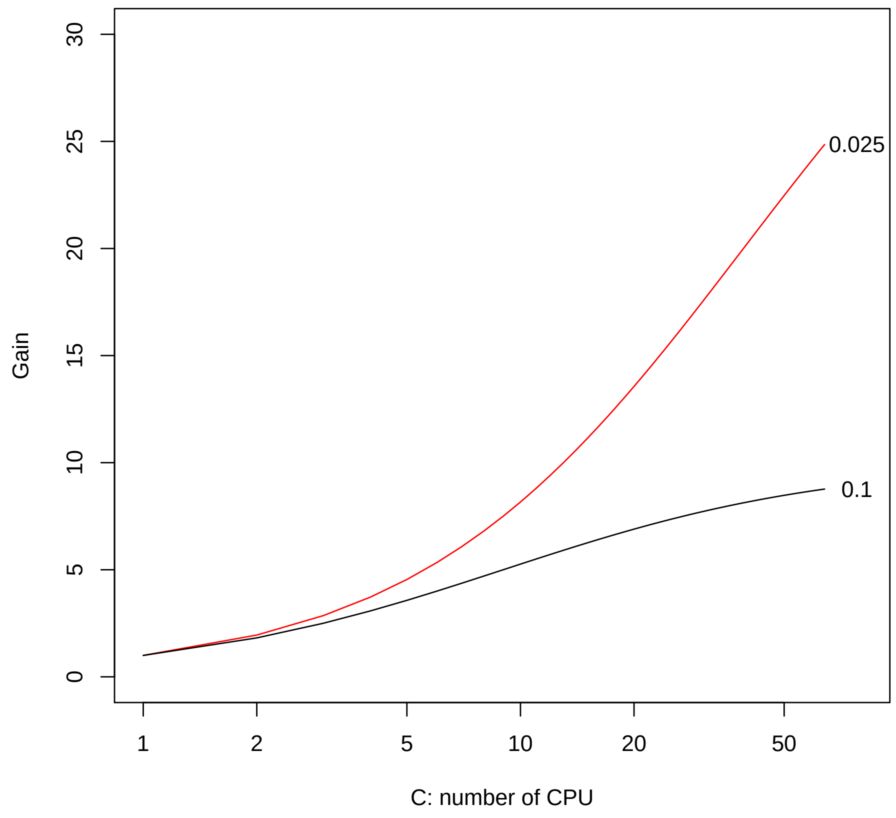

Figure 2: The theoretical gain due to multi-threading can be calculated by Amdahl's Law. Let b be the burn-in fraction and C be the number of threads, then the gain limit is $((1-b)/C+b)^{-1}$. In this figure, the theoretical gains are on the y-axis and the number of CPUs, the x-axis, for two settings: $b \in \{0.025, 0.1\}$.

Affiliation:

Rodney Sparapani rsparapa@mcw.edu
Division of Biostatistics, Institute for Health and Equity
Medical College of Wisconsin, Milwaukee campus