A parallel Random Forest implementation for R

Lawrence Mitchell*

11 January 2011

Abstract

We present the results of a project working on SPRINT to implement a parallel version of the random forest classification algorithm for R. We have chosen a task parallel approach which, on typical data sets, shows good strong scaling on the HECToR supercomputing facility to 128 processes with a speedup of 50 over the existing serial code. Our study also indicates where further work could be carried out to improve performance for large datasets.

Contents

*lawrence.mitchell@ed.ac.uk

1 Introduction

This dCSE project implements a parallel version of the random forest classifier (Breiman, 2001) for use in microarray analysis within the SPRINT framework, enabling usage on the HECToR supercomputing facility and other parallel machines.

R (R Development Core Team, 2010) is an open source software package for statistical computing and graphics. The majority of the implementation is in the R programming language. R is an interpreted, scheme-like, garbage-collected language. The core of R consists of the interpreter for this language, written in C, along with the garbage collector. Additionally, a number of core functions are written in C for performance reasons. R provides a foreign function interface for both C and Fortran code. The C interface is richer and allows access to R objects directly as well as the ability to call back into R.

SPRINT (Hill et al., 2008), the Simple Parallel R Interface is a project of the Division of Pathway Medicine and EPCC at The University of Edinburgh to provide parallelised workflows for microarray analysis within R. The aim is to provide functionality that is as close to existing code as possible so that end users, typically biostatisticians, can make use of HPC resources without altering their workflows excessively.

2 Tree classifiers

Given some data, we may wish to construct a predictive model which maps data observations to some conclusions about the data. For example, we may wish to classify days of the year into seasons based on observed weather data. One way of doing this is decision tree learning (Kotsiantis, 2007). We use data to construct a decision tree that makes predictions about the data. These trees are constructed using a set of *training data* (for which the data classification is known) and are then used to make predictions of the classification of unseen data.

2.1 The training data

Microarray data consists of measurements of expression levels for a large (typically thousands to hundreds of thousands) number of genes for a number of different individuals who are either in a control group or an "infected" group. In constructing a tree classifier, we aim to build a model that predicts the group of an unseen individual by looking solely at the measured expression levels of their genes. With this model we might then predict whether

an individual is susceptible to some disease.

Our training data consists of a number of data entries $d = \{A, c\}$ each of which is a set of attributes (gene expression levels) plus one entry describing the class $c \in C$ of the datum. Our tree classifier is a function $f : A \mapsto C$. We can then classify a further datum $e = \{A\}$ that is from an unknown class by applying the function f to its attributes.

2.2 Random forests

The random forest algorithm is an *ensemble* tree classifier that constructs a forest of classification trees from bootstrap samples of a dataset. The classification of an unseen datum is the modal class of the datum over all trees in the forest. The random forest algorithm, and the parallelised version described in this document, apply to any type of high-dimensional data as long as the input data are a simple numeric variables-by-cases matrix.

Random forests can be used to classify both categorical and continuous variables. Microarray data is always continuous, since gene expression levels are measured as a real number, however our implementation works in both the categorical and continuous cases.

The classification method is a popular one for microarray data since it does not just give the classification of a datum but also estimates which genes are important, builds an unbiased estimate of the classification error and does not overfit with increasing forest size (Breiman, 2001).

2.3 Building a single classification tree

The construction of a decision tree is conceptually straightforward. The tree consists of *nodes* each of which splits the dataset based on the value of some attribute. From the root node (which contains the full dataset) we recursively split into children until some terminating criterion is reached. This might be that the number of cases at a particular node drops below a threshold value, or that all cases share the same class. Such a terminal node is a called a *leaf node*. Once the tree is constructed we can classify an unseen case by sending it down the tree from the root node. The predicted class is the leaf node it ends up in.

The splitting criterion at a node is commonly to minimize the *Gini index* of the split (Breiman, 1984). This is a measure of how well the split does in reducing the mixing of classes in the child nodes: a good split will do a better job of reducing the mixing than a bad split. An alternative to the Gini

index is to pick the split which minimises the information entropy (Quinlan, 1986; Kotsiantis, 2007).

2.4 Choosing variables to split on

Most decision tree classifiers are deterministic. That is, for a particular dataset they will always produce the same tree. The trees grown by the random forest algorithm do not have this characteristic. This is due to the way variables are picked to split on at each node. Rather than choosing the best split amongst all variables at each node, a fixed size random subset of all the variables is chosen at each node. Hence an identical datum may be classified differently by two trees grown on the same dataset.

2.5 Data parallel tree generation

A number of papers describe data parallel approaches to the generation of decision trees (Joshi et al., 1998a,b; Shafer et al., 1996). All these approaches are designed around the types of data one encounters in the social sciences. That is, a very large number of cases (hundreds of thousands or millions) but only a small number of variables (tens or hundreds) per case. The parallelisation strategy is then to divide up the cases equally between parallel processes.

These algorithms do not map well onto microarray data, since the amount of parallelism across cases is low (there are only few cases). It would be possible to parallelise across the variables rather than the cases, however getting good load balance in random forest generation is tricky. We only split on a subset of the variables at each node and so some parallel processes will have less work than others at each split.

As a result of these issues, and since the typical microarray dataset is easily small enough to fit in the memory of a single R process (Forster, 2010), we decided not to pursue a data parallel approach. Instead we have implemented a task parallel random forest generator for SPRINT.

3 Growing a random forest

To construct a random forest, we generate some large number (typically thousands) of bootstrap samples of our original dataset and grow a classification tree for each of these samples. The data in the original dataset are then classified by sending them down each of these trees and selecting the

modal class. That is if 100 trees vote that case A is infected, while 900 trees vote that it is a control, the classification is as a control.

Generation of trees from each of the bootstrap samples can be carried out independently with the results combined at the end. As such we can naturally formulate a task parallel version of the algorithm by distributing the bootstrap samples amongst parallel processes and combining results at the end.

4 A task parallel implementation

Having decided on a task parallel implementation, and since one of the aims of the SPRINT project is to provide a drop in parallel replacement for a serial code, we decided to reuse the existing R code for serial random forest generation. Our implementation therefore exactly mimics the calling conventions and results of the serial code.

4.1 Distributing the work

A typical serial call to generate a random forest in R is something like

```
library(randomForest)
# data contains N rows each with M variables
# classes contains N entries, each one corresponding to the
# classification of a row of data
rf <- randomForest(x=data, y=classes, ntree=5000, ...)
# do further analysis of data
```

The parallelism here is available in the `ntree` argument which tells us how many trees the forest will contain. Our task parallel approach divides up the trees equally between available processes, generates these subforests in parallel and then combines them into a larger forest which is returned on the master process. The R interface is almost completely unchanged, the script now looks like

```
library(randomForest)
# load parallel library
library(sprint)
# replace randomForest with prandomForest
rf <- prandomForest(x=data, y=classes, ntree=5000, ...)
# do further analysis of data then exit MPI.
pterminate()
```

4.1.1 Sending R objects over MPI

R's data structures are not simple chunks of memory that can be easily sent to and fro over MPI. The basic datatype is a `SEXP`, which is a pointer to a struct

```
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} *SEXP;
```

Depending on the tag bit in the object header, the offset of the data can vary. Furthermore, to obtain all the data in (say) the `listsxp_struct` type, we need to walk the list. Hence, we cannot do something like:

```
SEXP foo = ...; /* Initialize foo somehow */
MPI_Send(&foo, sizeof(SEXP), MPI_BYTE, ...);
```

Fortunately, R has a serialisation mechanism for writing and restoring objects to disk between sessions. Although there is no public C interface to this feature, we can access it through R. The serialised object is an array of bytes with a header. To exchange the information, we just need to send the raw bytes and unserialize at the receiving end. So to send some data, we use

```
SEXP foo = ...; /* Initialize foo somehow */
SEXP bar = serialize_form(foo);
int length = length(bar);
MPI_Send(&length, 1, MPI_INT, ...);
MPI_Send(RAW(bar), length, MPI_BYTE, ...);
```

On the receiving side, we just do the inverse

```
SEXP foo;
SEXP bar;
```

```
int length;
MPI_Recv(&length, 1, MPI_INT, ...);
/* Allocate space for length bytes in bar */
MPI_Recv(RAW(bar), length, MPI_BYTE, ...);
foo = unserialize_form(bar);
```

4.2 Combining the results

Subforests generated on each worker process are combined at the end of the run into one large forest that is returned on the master process for further serial analysis. Our first implementation of this used a simple linear algorithm. We gathered all the data onto the master process and combined it there. Benchmarking this algorithm on more than 32 processes demonstrated that a significant amount of time was spent in the combination stage, limiting scalability. Interestingly, sending the data was not the bottleneck, but rather the linear complexity algorithm used to combine the results.

The combine operator is associative, so the most natural way to do this in an MPI environment is the `MPI_Reduce` function. Unfortunately, this does not work for our purposes. The object we wish to combine changes size at each level, which means we do not know what value to give as the `COUNT` argument to `MPI_Reduce`. Instead, we have written our own tree reduction code that applies the combine operator in parallel. Our implementation is sufficiently general to be usable in other parts of SPRINT where appropriate. We use a function signature similar to that of MPI, although the `count` and `datatype` arguments are unnecessary, since R objects are tagged with a type.

```
void reduce(SEXP in, SEXP *out, SEXP (*combine_fn)(SEXP, SEXP),
            int root, MPI_Comm comm)
```

4.3 Comparison with previous work

There are a number of existing task parallel implementations of the random forest algorithm. For example (Topić et al., 2005; Schwarz et al., 2010) both describe stand-alone programs for random forest classification. Additionally, the `foreach` package (REvolution Computing, 2009) can be used to generate random forest in parallel, although the user has to do more work by hand than in our script. None of these parallel implementations carries out combination of results in parallel, instead they all use a gather-to-master approach. As we show later, for our analysis using a serial gather-to-master approach introduced a bottleneck that a tree-reduction has avoided.

5 Performance results

We report performance results of our implementation on both the XT4 and XT6 incarnations of HECToR. The former consists of quadcore nodes, the latter of 24 core nodes as four hexcore dies. We find the performance does not vary significantly between the systems.

5.1 Datasets used

We have tested our code on a microarray dataset containing 62 individual cases each described by 23292 genes. This is a typical sized microarray data. We have also constructed larger datasets by replicating (with noise) the rows or columns of the data.

5.2 Comparison of combination algorithms

We first present the timing behaviour of the two different combination algorithms described in section 4.2. Recall that the initial implementation used a simple linear approach combining all subforests on the master process. Our final implementation uses a tree reduction that carries out the combination of results in logarithmic rather than linear complexity. Figure 1 shows the walltime for generating a random forest of 5000 trees using both linear and tree-reduction algorithms for combining the data.

5.3 Memory contention

When running in parallel on a multicore node we observe a severe performance degradation when processes share the same memory bus, even if there is no communication of results at all. This is a particular issue for the XT6 incarnation of HECToR with 24 cores per node. When running multiple instances of the serial code simultaneously on a single node, we see a significant degradation in the performance. The 24 cores of an XT6 node are divided into four hexcore dies. As long as only a single process is placed on each die, performance does not suffer. However, when we place multiple processes on a die, we see a drop in performance. When two processes are placed on each die, time to solution increases by 7%, with four processes this rises to 13% and if the dies are packed (24 processes on a node) we experience a 17% drop in performance over a single process. This suggests it may be worthwhile to tune the memory access patterns and serial performance of the random forest code before investing more time in parallel scaling.

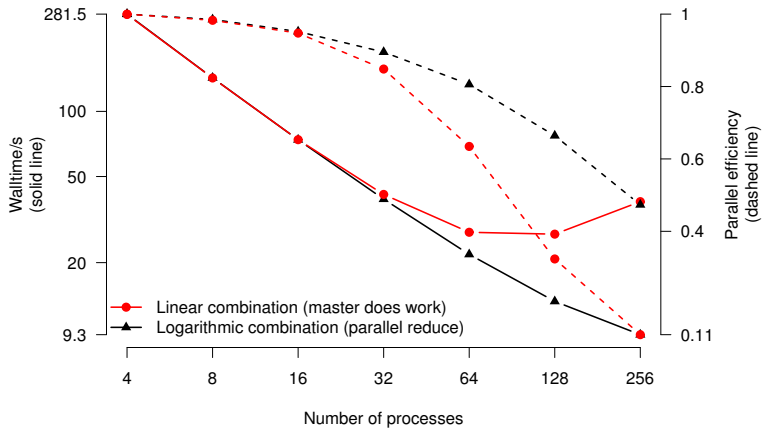

Figure 1: Comparison of the effects of logarithmic and linear forest combination algorithms. The data used here have 23292 genes and 248 cases. Note that the code using a serial combination of results on the master only scales to 64 processes, whereas when combining in parallel allows good scaling to 256 processes.

5.4 Generating forests faster

Microarray data from a given experiment is a fixed size. As such, the parallel scaling performance we are most interested in is strong scaling. That is, faster time to solution for a fixed size input with an increasing number of parallel processes. Modulo memory contention issues (see section 5.3), reducing the number of trees in a forest decreases the time to generate the forest linearly. Figure 2 shows the linearly increasing time to solution with increasing forest size.

This means that if we can balance the trees to be generated equally across all processes, we will get good strong scaling behaviour of the forest generation step. To obtain good overall scaling, we then need to ensure that the communication and computational overheads of broadcasting the data and combining the results are low. Broadcasting the data from the master process can be carried out with `MPI_Bcast`. We therefore rely on the vendor's MPI implementation doing this efficiently. Our approach for combining results has already been described in section 5.2.

Figure 3 shows the scaling behaviour of our code when solving a fixed size problem on an increasing number of parallel processes. We see good parallel efficiency up to 64 processes, giving a speedup of around 40 over

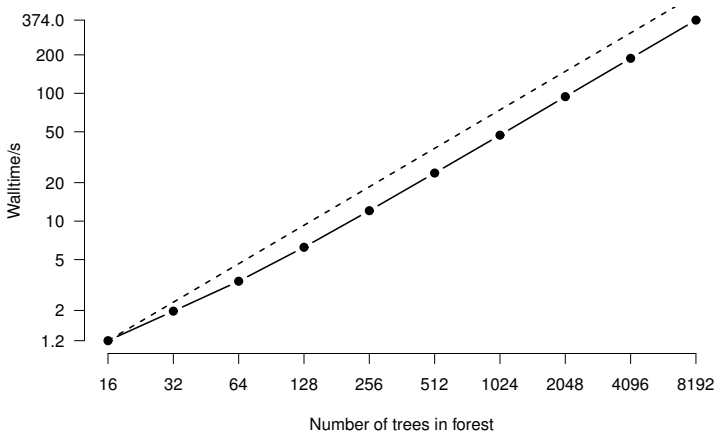

Figure 2: Time to solution for serial generation of random forests as the forest size increases. Dashed line shows expected timings if doubling forest size exactly doubled execution time, indicating that we will get good parallel speedup if we can distribute forest generation between processes.

the same serial code on HECToR. Results are essentially identical on both the XT4 and XT6. On 64 processes, generation of this random forest takes approximately 9 seconds, rather than 390 seconds in serial.

5.5 Weak scaling in forest size

If we increase the size of the forest, but wish to generate the result in the same amount of time as previously we can do so by increasing the number of parallel processes. If our implementation has good *weak scaling* characteristics then (as shown in figure 2) when doubling the number of trees in the forest we should be able to double the number of parallel processes and obtain results just as quickly. Figure 4 shows the parallel efficiency of our implementation when increasing the size of the forest. Recall (section 5.3) that a significant proportion of the inefficiency comes from poor memory access performance. Even without any communication overheads, contention for memory bandwidth on the local nodes causes a slowdown when running in parallel.

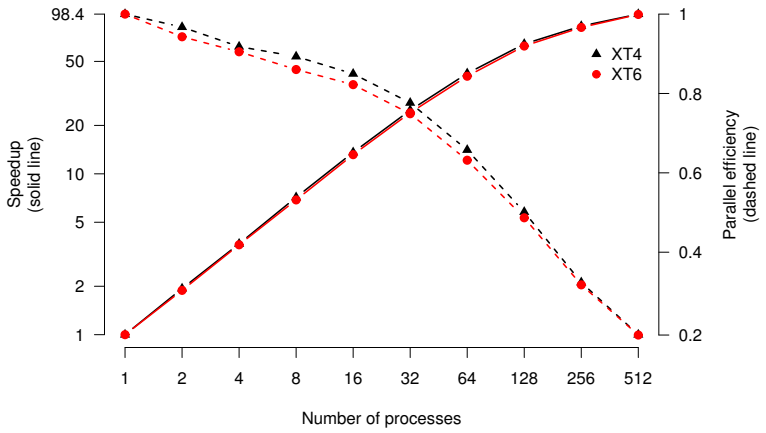

Figure 3: Parallel speedup (solid) and efficiency (dashed line) in the generation of a random forest with 8192 trees on both the XT4 and XT6

5.6 Weak scaling in dataset size

There are two different dimensions in which we can increase the size of the dataset. Either the number of cases we're considering or the number variables measured. The former is experimentally expensive in microarray data (it's an additional person). The latter is less expensive, it corresponds to measuring the expression levels of more genes.

Increasing the number of variables per case results in a linear increase in time to solution[1]. On a system with 4GB of RAM we can run a serial analysis with a dataset of around 700MB (1.5 million genes and 62 cases, see figure 5). Larger than this and the system starts swapping, leading to a serious performance drop off.

If we increase the number of cases, but hold the number of variables constant, then the time to solution increases faster than linearly. As a result, if we double the number of cases, we need to more than double the number of parallel processes to compensate.

6 Conclusions

We have implemented a task parallel version of the random forest algorithm for use in R using the SPRINT library. For typical use cases, we can ob-

[1] As long as all the data structures fit in RAM

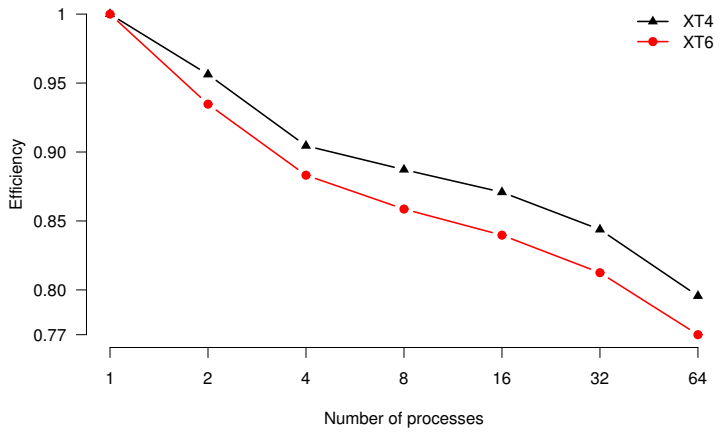

Figure 4: Parallel efficiency with a constant number of trees per parallel process. An ideal implementation would have a parallel efficiency of one for all job sizes

tain a speedup of around 40 over the same serial code on HECToR. Rather than implementing the algorithm from scratch, we used an existing serial implementation and added a parallel wrapper around it. Unlike other task parallel packages for R, we also implemented a tree-reduction algorithm to combine results in parallel rather than serial. This had a surprisingly large effect on the overall performance. Our interface exactly mimics the existing serial implementation: modifying existing serial R scripts to take advantage of this functionality is trivial.

Our implementation shows both good strong and weak scaling characteristics, but has demonstrated a number of areas where the existing serial implementation for R is suboptimal. A future useful area of work may be to incorporate a better serial library for random forest generation into R.

Development of the code has been carried out in the main SPRINT repository. It is already available to developers and will ship as an R package in a forthcoming release of SPRINT.

6.1 Future work

6.1.1 Serial performance

As previously mentioned, the serial random forest implementation available in R has been written with data from the social sciences in mind: large

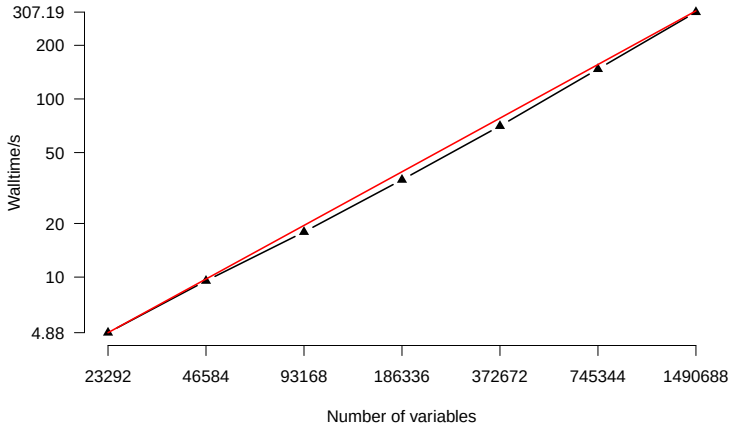

Figure 5: Effect of increasing the number of variables per case on the serial time to solution. Red curve shows linear time increase, indicating that increasing the number of processes along with the data should give a a constant time to solution.

numbers of cases and few variables per case. This is the exact opposite of the type of data seen in microarray analysis. As a result, the code is not very efficient when dealing with microarray data, especially as the number of variables becomes very large.

These issues with large microarray data have previously been reported in (Ziegler et al., 2007; Schwarz et al., 2007). The random jungle package (Schwarz et al., 2010) has been expressly written to avoid these problems. At present, no R interface exists to this new package. If such an interface were developed we could usefully leverage the better performance within our parallel framework.

6.1.2 Data parallelism

Our implementation cannot handle huge datasets, since we are limited by the memory in a single R process. However, as discussed in section 2.5, data parallel algorithms for generating classification trees do exist. It should be possible to adapt these for use with typical microarray data given some time and thought. In addition, the random jungle library claims to be much more memory efficient (Schwarz et al., 2010) than current R implementations of random forest which should allow for larger datasets to be analysed without resorting to a data parallel approach.

7 Acknowledgements

The SPRINT project is funded by a Wellcome Trust grant [086696/Z/08/Z]. Data were kindly provided by Thorsten Forster of the Division of Pathway Medicine, University of Edinburgh.

This project was funded under the HECToR distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR – A Research Councils UK High End Computing Service – is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE support Service is provided by NAG Ltd. http://www.hector.ac.uk.

References

L. Breiman. *Classification and regression trees*. Wadsworth, Belmont, California, 1984.

L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

T. Forster. Personal communication, 2010.

J. Hill, M. Hambley, T. Forster, M. Mewissen, T. Sloan, F. Scharinger, A. Trew, and P. Ghazal. SPRINT: A new parallel framework for R. *BMC Bioinformatics*, 9(1):558, 2008.

M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the International Parallel Processing Symposium*, pages 573–579, 1998a.

M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. Technical report, University of Minnesota, 1998b. URL `http://glaros.dtc.umn.edu/gkhome/fetch/papers/scalparc.pdf`.

S. B. Kotsiantis. Supervised machine learning: a review of classification techniques. *Informatica*, 31:249–268, 2007.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

REvolution Computing. *foreach: Foreach looping construct for R*, 2009. URL `http://CRAN.R-project.org/package=foreach`. R package version 1.3.0.

D. Schwarz, S. Szymczak, A. Ziegler, and I. Konig. Picking single-nucleotide polymorphisms in forests. *BMC Proceedings*, 1(Suppl 1):S59, 2007.

D. F. Schwarz, I. R. König, and A. Ziegler. On safari to Random Jungle: a fast implementation of Random Forests for high-dimensional data. *Bioinformatics*, 26(14):1752–1758, 2010.

J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 544–555, 1996.

G. Topić, T. Šmuc, Z. Šojat, and K. Skala. Reimplementation of the random forest algorithm. In M. Vajteršic, R. Trobec, and P. Zinterhof, editors, *Parallel Numerics '05*, pages 119–125, 2005.

A. Ziegler, A. L. DeStefano, and I. R. König. Data mining, neural nets, trees – problems 2 and 3 of genetic analysis workshop 15. *Genetic Epidemiology*, 31(Supplement 1):S51–S60, 2007.