Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria ISSN 1609-395X
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Statistical Computing and Databases: Distributed Computing Near the Data

Fei Chen **Brian D. Ripley**

Abstract

This paper addresses the following question: "how do we fit statistical models efficiently with very large data sets that reside in databases?"

Nowadays it is quite common to we encounter a situation where a very large data set is stored in a database, yet the statistical analysis is performed with a separate piece of software such as R. Usually it does not make much sense and in some cases it may not even be possible to move the data from the database manager into the statistical software in order to complete a statistical calculation.

For this reason we discuss and implement the concept of "computing near the data". To reduce the amount of data that needs to be transferred, we should perform as many operations as possible at the location where the data resides, and only communicate the reduced summary information across to the statistical system.

We present details of implementation on embedding an R process inside a database (MySQL) and remotely controlling this process with another R session via communication interfaces provided by CORBA. In order to make such distributed computing transparent to the end user, we discuss modifying the R engine to allow computation with external pointers such that an expression involving an external pointer reference is evaluated remotely at the location where the object pointed to resides. In addition, we implement distributed R data frames to give R the ability to handle very large data sets with PVM and ScaLAPACK.

1 Introduction

The advent of automatic data collection proves a challenge to statisticians responsible for analyzing the collected data. A much quoted example is NASA's Earth Observing System of orbiting satellites sending one terabyte of data to a receiving station every day (Kahn and Braverman, 1999). On a smaller scale, the State

Farm Insurance Company in the U.S.A has more than 66 million insurance policies (http://www.statefarm.com). Assuming ten categorical variables are collected for each policy and one byte is used to store each variable, this amounts to more than half a gigabyte of data, and the derived design matrix will probably be even bigger. Although 1Gb memory is not uncommon these days, it is still a taxing job for R to fit models such as the Generalized Linear Model on data sets of this size.

A more limiting factor probably is network bandwidth. Large data sets tend to be stored in databases. To fit a model using R usually means that the data has to be transferred across a network into R. As a result, computation time is dominated by the overhead of data transfer for large data sets.

This paper addresses the following question: "how do we fit statistical models efficiently with very large data sets that reside in databases?" To answer this question, we propose the concept of "computing near the data". To reduce the amount of data that needs to be transferred, we should perform as many operations as possible at the location where the data resides, and only communicate the reduced summary information across to the statistical system.

Temple Lang (2000) provides one solution to the data transfer problem by embedding the R engine inside the database. Using his example, embedding allows us to write `SQL` statements such as

```
select f(x) from table
```

where data is transformed first on the database side and then retrieved into R, instead of R statements like

```
f(select x from table)
```

which implies transferring data from the database into R before any analysis takes place.

Based on this embedding framework, we propose to build a computing environment for the analysis of large data sets. The basic idea is to treat every database table as a data frame (*cf.* `Rdbi.PgSQL` (http://rdbi.sourceforge.net))[1]. The database is attachable by R to its search path. Upon being attached, names of tables residing in the database become external pointers in R. Any operation involving these external pointers is executed on the database side, using the embedded R engine. In this setup the R user can bypass `SQL` statements and only program in one language, S. There will still be data transfer from the database into embedded R, but the overhead is minimized because we assume the embedded R runs on the same machine as the database. Last but not least, we implement distributed R data frames using PVM to handle very large data sets transparently.

2 Computing near the data

Most statistical packages today are monolithic systems. The statistical software alone is responsible for all tasks related to data analysis, from data management to data visualization. As data set size grows, it is sometimes difficult to extend these

[1]Relational databases usually store tables in normalized forms. We do not discuss automatic de-normalization in this paper. Instead we just assume tables are stored de-normalized, as data frames are.

systems to meet the new computational needs. One limitation of the monolithic design is that due to its tight integration it provides limited flexibility in the evolution toward a component-based system. For example, if the volume of data exceeds the built-in data management capabilities, it may not be possible simply to upgrade the data management functionality alone without taking apart the entire system.

The alternative is to take as much out of the monolithic system as possible and to design a modular system where different tasks run as separate processes on different processors (servers) outside the statistical package (the statistics server). In this scenario, the statistics server's job is to handle message passing and to manage the overall flow of an algorithm execution.

In fitting a statistical model, we often find that there are two types of computations, each involving different amounts of data. One type requires the entire data set; the operations performed on the data tend to be primitive, usually involving some sort of summarizing operation. The other type requires more complicated manipulation of summary results, but the amount of data required is small.

Generally speaking, in optimization we want to maximize some objective function,

$$\max_{\beta} \mathcal{L}(\beta, X),$$

that depends on some parameter vector β and some data X. This is usually solved iteratively, and at time step $t+1$ we use an iteration

$$\beta^{(t+1)} = \beta^{(t)} + U(X, \beta^{(t)}).$$

Here U is a correction factor defined by a particular algorithm and the objective function $\mathcal{L}(\cdot)$. The data-intensive part of the algorithm lies in calculating U. In all but the simplest algorithms, the calculation involves multiple steps; some of them require X, some of them are just a simple update of intermediate results. The parts that require X are often fairly straightforward such as computing the objective function or its derivatives. For example, the popular Quasi-Newton method with BFGS update has the form

$$\beta^{(t+1)} = \beta^{(t)} + \left(-\frac{\beta^{(t)} S S^T \beta^{(t)}}{S^T \beta^{(t)} S} - \frac{G G^T}{G^T S} \right),$$

where $S = \beta^{(t+1)} - \beta^{(t)}$ and $G = \boxed{\mathcal{L}'(X, \beta^{(t+1)})} - \boxed{\mathcal{L}'(X, \beta^{(t)})}$. As we can see from the framed boxes, the data-intensive part of the algorithm is straightforward (if the derivative is known).

This observation suggests a distributed computing idea where we perform the data-intensive but algorithmically less sophisticated operations on the entire data set near where this data set is located (as a kind of building block of a more sophisticated algorithm). We send the summary results (such as the value of an objective function or its derivatives) back to the server responsible for the algorithmic flow. This server then in turn carries out more complicated actions such as constructing the correction factor U, inverting a matrix, or computing a linear interpolation. In other words, we break computation into pieces. Some pieces are carried out on the database side while others are carried out inside a statistics server.

3 System design and implementation

In our proposed model, we adopt a component-based design where the responsibilities of model fitting, data management and computation are delegated to different processors. We define these three tasks by implementation. A database system is responsible for managing data, including creating model matrices. Low level linear algebra routines are computational tasks. Communicating model formulas, managing algorithm flows as well as inter-system communications during algorithm execution are part of model fitting.

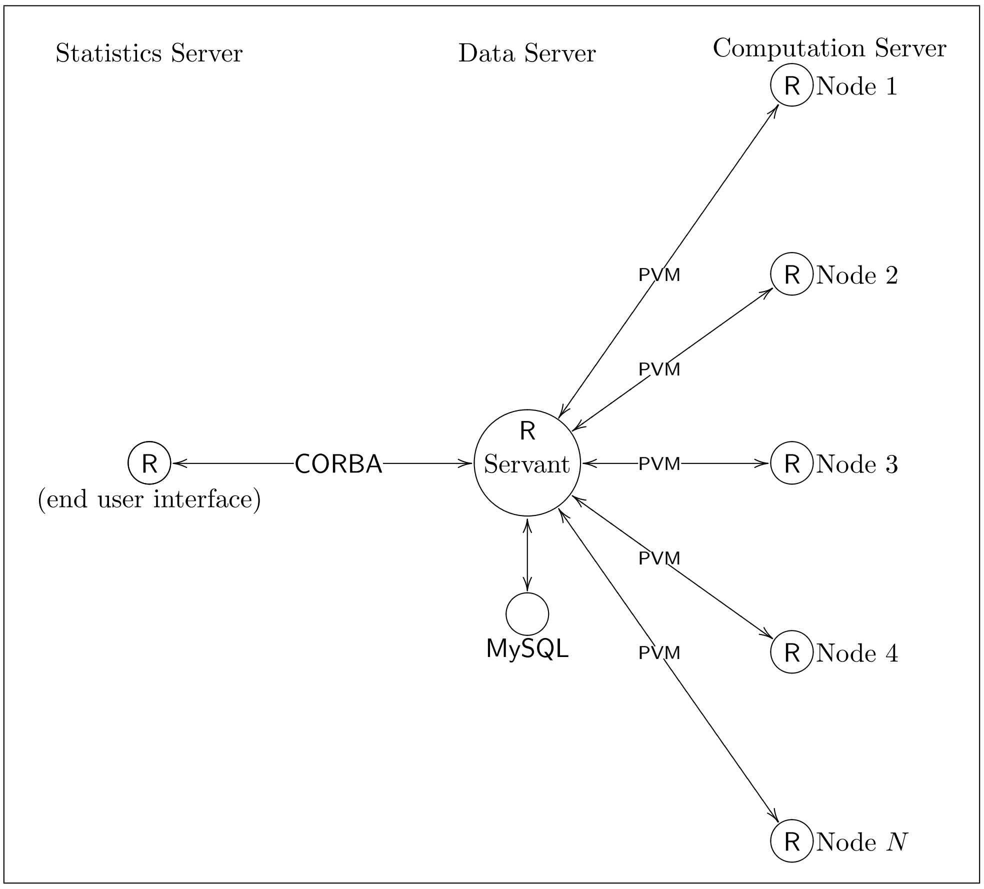

Figure 1: Computing near the data. This figure illustrates the idea and implementation of performing computation where the data resides. The end user interface is accessible through the Statistics Server, R, which sends out job requests for statistical analysis via a CORBA interface. The execution of an analysis actually takes place on the data server, MySQL, through a CORBA servant running embedded R. The servant thus understands instructions sent by the statistics server and is able to execute any R code. The computing server, a cluster of processors each also running an embedded R process, is available to the servant to speed up computation using PVM running on a N-node process grid.

Figure 1 illustrates the idea and implementation of performing computation where the data resides, e.g. in a database management system. The system consists of three main parts: a statistics server, a data server and a computing server. The computation server is a collection of processors awaiting jobs submitted from the central server-the statistics server. The statistics server is essentially an overarching "operating system", which, by communicating with a server that embeds R, dictates how the data server and the computation server are to interact with each other and how the computation server is to finish a computational task.

The database engine used throughout this paper is MySQL (Axmark, Widenius, Cole, Lentz, and DuBois, 2002). Communications between R processes are provided by CORBA (Object Management Group, 1996). The computation server is implemented using PVM (Geist, Beguelin, Dongarra, Jiang, Manchek, and Sunderam, 1994) and makes calls to ScaLAPACK (Blackford, Choi, Cleary, D'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker, and Whale, 1997). See Figure 1.

The requirements that we have of a distributed computing system are simple. First, we would like to assume that the end users operate in only one language, S, and thus avoid having to issue SQL-like statements such as `select x from X` from within R. Second, we would like to have transparency of computation from the perspective of the end user; that is, we would like distributed computing to take place automatically as much as possible. Third, we need a system that can handle very large data sets.

We use external pointer references (Tierney, 2002) extensively to give end users a transparent interface to distributed computing. An external pointer is simply a memory address that points to anything external to the session, for example, database tables, CORBA objects, etc. R uses this mechanism to refer to and access external objects.

3.1 Single-language requirement: attaching the remote R search path

In order to hide details of SQL from the end user, we decided that tables in databases should behave like data frames. This means we first need to have a method in R to refer to these tables. Our solution is to implement methods to attach MySQL databases as well as search paths of other (remote) R sessions. In the first case, all data tables residing in the database are declared as external pointers. Table names become R external pointer names. For example,

```
R> # initialize a database connection
R> db <- init.mysql(user="feic",database="insurance")
R> # attach the database
R> attach.mysql(db)
R> search()
[1] ".GlobalEnv" "package:ctest" "Autoloads" "package:base"
[5] "database:mysql"
R> ls(database:mysql)
[1] X Y Z
R> typeof(X)
[1] "externalptr"
R> class(X)
[1] "mysql"
```

In the second case, all variables in a remote R search path become external pointers with the same names in the current local R session. Attaching a remote R search path is implemented using the CORBA communications interface.

```
R> # initialize a corba connection to the remote R session
R> corba <- init.corba()
R> # see what the remote R has attached
R> eval.corba("search()",corba)
[1] ".GlobalEnv" "package:ctest" "Autoloads" "package:base"
R> # now attach the remote R's mysql search path
R> attach.corba(".GlobalEnv",corba)
R> # check out what we've attached
R> search()
[1] ".GlobalEnv" "package:ctest" "Autoloads" "package:base"
[5] "corba:.GlobalEnv"
R> ls("corba:.GlobalEnv")
[1] a b c
R> typeof(a)
[1] "externalptr"
R> class(a)
[1] "corba"
```

3.2 Transparency requirement: computing with external pointers

Making references from R to database tables is the first step toward transparent distributed computing. The next functionality we need to provide is to make external pointers behave as if they were R data frame objects.

The structure declaration `R_ObjectTable` in `include/R_ext/Callbacks.h` is R's built-in mechanism to handle external pointers transparently. It provides a set of hooks to which end users can attach their own methods for reading, writing and attaching external objects. However it assumes that computations take place locally. That is, an externally pointed object has to be read into R's main memory before computations involving it can take place. Therefore, if we attach a database directly onto the user's current R session with `R_ObjectTable`, it would mean transferring data from database tables into R. This is precisely the type of overhead we wish to avoid.

Our solution is to start an R process on the same computer where the database resides. This R, being "closer" to the data than the end user's current R session, is actually the process that attaches the database. This remote R uses the `R_ObjectTable` mechanism to access and compute with external pointers as if they were R data frames.

The end user's current R session, considered to be the "statistics server" in Figure 1, does not directly attach the database (to avoid data transfer). Instead it attaches the search path of the "remote" R running near the database that has actually attached the database. CORBA is used to provide a communication interface between the two R sessions.

That is, the current (user) R session does the following:

```
R> # initialize a corba connection to the remote R session
```

```
R> corba <- init.corba()
R> # tell remote R to attach the database
R> eval.corba("db<-init.mysql",corba)
R> eval.corba("attach.mysql(db)",corba)
R> # see what the remote R has attached
R> eval.corba("search()", corba)
[1] ".GlobalEnv" "package:ctest" "Autoloads" "package:base"
[5] "database:mysql"
R> # now attach the remote R's mysql search path
R> attach.corba("database:mysql",corba)
R> # check out what we've attached
R> search()
[1] ".GlobalEnv" "package:ctest" "Autoloads" "package:base"
[5] "corba:database:mysql"
R> ls(corba:database:mysql)
[1] X Y Z
R> typeof(X)
[1] "externalptr"
R> class(X)
[1] "corba"
```

Note that although the names of tables persist, the class of these external pointers are `corba` but no longer `mysql`.

This is a little convoluted. But with a few modifications to the R evaluator, we achieve transparent distributed computing. Whenever an external pointer reference is used in an expression, and if its class is `corba`, this expression, along with its enclosing environment, is shipped to the remote R session for evaluation.

```
R> ls(corba:database:mysql)
[1] X Y Z
R> X[[1]]
CORBA: executing "X[[1]]" and returning
[1]  1  2  3  4  5  6  7  8  9 10
R> lm(x~y,data=X)
CORBA: executing "lm(x~y,data=X)" and returning
...
```

Modifying the R evaluator

The modifications to R's evaluator mainly consist of three parts. First an extra field is inserted in the structure declaration of `Defn.h:FUNTAB`. This is a flag indicating whether the corresponding routine can be executed remotely or not. For example, the R operator `[[` is declared to be remotely executable, hence the expression above `X[[1]]` is sent to the remote R session by the current R evaluator and the appropriate subset is returned. One useful `TRUE` remote declaration applies to the operator {. The way the evaluator works assures all statements inside the {} block are executed remotely, if an external pointer is found somewhere inside the block.

For function closures, an equivalent `remote` attribute is set to indicate the same. For example, `rm(X)` is interpreted to mean remove `X` locally if `attr(rm,"remote")==FALSE`, otherwise it means remove `X` from the remote R session. Thus in the previous example, `attr(lm,"remote")` has to be set to `TRUE` or an error would have been encountered.

The second modification is the addition of a check for external pointers in the R evaluator (`eval.c:eval`). Currently this check simply traverses the tail part of the current operation (in the parse tree) and looks for external pointers in the enclosing environment. If any external pointer is found, the current operation is then considered for remote execution.

The third modification alters the behavior of the R evaluator when an expression contains external pointers is encountered. In R, computations take place when

```
PRIMFUN(op)(e, op, CDR(e), rho)
```

is called. Correspondingly, for any remote execution we implemented a routine

```
remote_PRIMFUN(e, op, CDR(e), rho)
```

This uses CORBA to communicate between the remote R processes. The CORBA servant interface is

```
module R{
 interface EvalR
 {
  void initialize_R();
  void shutdown_R();
  void eval_R(in string s,
              in string rho,
              out string val);
  void call_PRIMFUN(in string e,
                    in string op,
                    in string args,
                    in string rho,
                    out string val);
 };
};
```

the important routine being `call_PRIMFUN`, which skeletally is

```
void EvalR_i::call_PRIMFUN(const char *e_str,
                           const char *op_str,
                           const char *args_str,
                           const char *rho_str,
                           CORBA::String_out result)
 {
  // unserialize arguments
  ...
  // evaluate expression e_str in environment rho_str
  ...
  // serialize result
  ...
  // copy serialized result into result
 }
```

With such a CORBA servant running on the remote R side, the current R session can invoke `call_PRIMFUN` with the appropriate CORBA object reference

```
SEXP remote_PRIMFUN(SEXP e, SEXP op, SEXP args, SEXP rho)
{
 R::EvalR_var R_ref; /* reference to remote R */
 // obtain reference
 ...
 // serialize arguments
 ...
 // invoke remote method
 R_ref->call_PRIMFUN(e_str, op_str, args_str, rho_str, result_str)
 // check for evaluation error remotely
 ...
 // return unserialized result_str
 ...
}
```

3.3 Large data set requirement: distributed R data frames

Parallel computing is a scalable way to solve the large data set problem. Using PVM, we implemented a distributed R data frame concept to take advantage of existing parallel computing software such as ScaLAPACK. The idea is simple. The remote R session (the one that controls the database) starts a process grid. Each process on this grid runs a separate embedded R. Embedding R lets us take advantage of the memory management and language parsing capabilities available in R. Each node runs an event loop, waiting for instructions (i.e. S expressions) from the master R process. Large data frames that normally would not fit into the memory of any one computer are broken up into blocks and distributed onto the process grid. Local portions of a global data frame are allocated, protected and garbage collected as any other local R objects. We require the name of a distributed data frame be the same across all processes on the grid, so we do not need distributed name space management. Finally, distributed data frames exist as long as the process grid exists. Thus data frames only need to be distributed once, and the resulting objects can be operated upon as many times as necessary without incurring further network overhead.

To distribute a database table onto a process grid, we do something like the following.

```
R> # start a 2x3 grid with block size 2x2
R> gridinfo<-init.grid(nprow=2,npcol=3,mb=2,nb=2)
R> # connect to mysql
R> db <- init.mysql()
R> # attach the database
R> attach.mysql(db);
R> ls(database:mysql)
[1] X
R> dim(X)
[1] 9 4
R> # distribute an external pointer as R object Xdist
R> Xdist <- distribute.extptr(X,gridinfo)
R> # Xdist is a regular matrix object, local portion of X
R> is.matrix(Xdist)
```

```
[1] TRUE
R> class(Xdist)
[1] "scalapack"
R> Xdist
     [,1]     [,2]
[1,]    1 1.000000
[2,]    1 0.500000
[3,]    1 0.200000
[4,]    1 0.166667
[5,]    1 0.111111
attr(,"class")
[1] "scalapack"
attr(,"desc")
dtype  ctxt     m     n    mb    nb  RSRC  CSRC   ldd
    1     0     9     4     2     2     0     0     5
This is local part of a distributed matrix
R> # now run qr decomposition, results stored in "XdistQR"
R> XdistQR <- qr.scalapack(Xdist,gridinfo)
R> # collect results
R> collect.scalapack(XdistQR,m=4,n=1,gridinfo) - qr.R(qr(X))
[1] 0 0 0 0
...
```

All distributed objects have a `scalapack` class, so we can easily use operator overloading to hide the details.

4 Example: fitting a GLM model distributedly

The motivation behind all this work was the desire to fit a Generalized Linear Model using a data set that resides in a database. The idea was to have the database perform the $X'WX$ and $X'WZ$ calculations, and to send the reduced summary results to R to obtain the final solution to the least squares problem, $(X'WX)^{-1}X'WZ$, and then repeat. Here X is the design matrix, Z the adjusted response, and W the weights.

Having implemented a more general distributed computing system, we now have the ability to do this. But instead of doing the explicit $X'WX$ calculation, we use the ScaLAPACK routine `pdgels` for solving least squares problems instead.

```
R> # initialize the CORBA object on the data server side
R> corba <- init.corba()
R> # initialize process grid for the data server
R> gridinfo<-init.grid(nprow=2,npcol=3,mb=2,nb=2)
R> # tell remote R to open a database connection
R> eval.corba("db <- init.mysql()",corba)
R> # tell remote R to attach the database
R> eval.corba("attach.mysql(db)",corba)
R> # attach the remote R search path
R> attach.corba("database:mysql",corba);
R> ls(corba:database:mysql)
[1] X Y
```

```
R> names(X)
CORBA: executing "names(X)" and returning
[1] "x1" "x2"
R> names(Y)
CORBA: executing "names(Y)" and returning
[1] "y"
R> # Poisson regression here
R> fam <- poisson()
R> # Y is external pointer, executed remotely, because "+" can be
R> # remotely evaluated
R> mustart <- Y + 0.1
CORBA: executing "Y + 0.1" and returning
R> # initializing glm
R> eta <- fam$linkfun(mustart)
R> mu <- fam$linkinv(eta)
R> mu.eta.val <- fam$mu.eta(eta)
R> # here ( is set to be not remotely evaluatable, but - is
R> z <- eta + (y - mu)/mu.eta.val
CORBA: executing "y - mu" and returning
R> # get the weights
R> w <- sqrt((mu.eta.val^2)/fam$var(mu)
R> # build the model matrix
R> mod <- terms(y~x1+x2)
R> # distribute the model matrix on the fly
R> Xmod <- distribute.modelmatrix(X,mod,gridinfo)
R> # distribute z
R> Z <- distribute.scalapack(z,gridinfo)
R> # w is diagnol matrix
R> W <- distribute.diag.scalapack(w,gridinfo)
R> # weight model matrix with weights
R> XWmod <- pdgemm.scalapack(W,Xmod,gridinfo) # sqrt(w)%*%X
R> # now W, Z, and XWmod are all distributed R objects
R> # so we can fit linear model using pdgels
R> fit <- lm.fit.scalapack(XWmod,Z,gridinfo)
R> # collect the coefficients back
R> beta <- collect.scalapack(fit,m=3,n=1)
... repeat til convergence
```

5 Discussion

One detail that we have not discussed is creating a model matrix on the fly, given a database table and a model formula. Data frames in R contain meta information (variable type, levels information, etc) on each column. In relational database management systems such information tend to be stored in different tables (i.e. tables are normalized). As such factor and levels information are not immediately available for every table we want to treat as a data frame. We assume the convention that there exists an entry in a table called `DataFrameVarInfo` that registers the variable type and levels information for each component of a table/data frame. Another table, called `DataFrameDimInfo`, contains a corresponding entry of the dimension of this data frame. In this fashion, variable and levels information is readily available, and a model matrix can be constructed on the fly without having to read the entire table into R memory first. This is what `distribute.modelmatrix` does.

In implementing remote evaluation of external pointers, there is the issue of the enclosing environment for an expression. Ideally we would find all the free identifiers in the expression to be remotely executed, create a new environment containing the values for these free identifiers, and evaluate the expression in this new environment on the remote R process. Such distributed scope is not currently implemented. Instead the entire enclosing environment of the expression in question is packaged up and shipped to the remote server, potentially causing a large overhead.

A related issue is lazy evaluation. If we have a function

```
R> rm(m)
R> test <- function(x, b=rnorm(10, mean=m))
     {
       return(x)
     }
R> test(10)
[1] 10
```

since `b` is never used within the function body, the expression `rnorm(10, mean=m)` is actually never evaluated. Thus even though `m` does not exist, `test` still returns a valid answer.

If `x` is an external pointer to a database table, we need to decide what variables to send to the data server. Should `b` be unevaluated and represented internally as an R type `PROMSXP`, which will end up generating an error when evaluated on the server side (if `b` were actually referred to in the function body, that is), or should `b` be evaluated in the current session first, which will correctly generate an "object not found error", but is contradictory to the principle of lazy evaluation?

6 Conclusions

This paper proposes the idea of "computing near the data". We envision a computing environment where the tasks of model fitting, data management and computation are delegated to different servers, and we suggest that data-intensive calculations take place on the database server side to minimize data transfer.

The implementation of such a component-based distributed statistical computing environment makes heavy use of the external pointer mechanism and the embedded R library. Embedding R frees us from transferring data before computations can take place, and allows us to create a distributed R data frame interface without implementing a distributed name space and memory management scheme. External pointers, on the other hand, give us the ability to refer to various objects external to an R session, and thus play an essential role in helping us keep track of different objects in a distributed setting.

So we end up with a system that has the general ability to handle very large data sets in a practical and scalable manner. From the end user's point of view, R has the transparent ability to compute with very large data frames, although behind the scenes these data frames may be far away from the computer on which the current R session is running. We now have the software infrastructure for "computing near the data". So it is possible to modify the `optim` code, for example, and to use it to perform maximum likelihood estimation over very large data sets (this is currently in the works). In the same vein, integrating PVM as part of this computing environment lets us take full advantage of parallel subroutine libraries already in existence, such as ScaLAPACK, enabling us to easily fit linear models, perform matrix decompositions, etc.

All the ideas described in this paper have been implemented and are currently working in an experimental setting using modified R source code, but much work remains to be done on providing a user-friendly interface.

Acknowledgements

Fei Chen is funded by EMB Consultancy, Epsom, England, UK.

References

David Axmark, Michael (Monty) Widenius, Jeremy Cole, Arjen Lentz, and Paul DuBois. *MySQL Reference Manual*, 2002. URL http://www.mysql.org.

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whale. *ScaLAPACK Users' Guide*. SIAM, 1997.

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA, 1994.

Ralph Kahn and Amy Braverman. What shall we do with the data we are expecting from upcoming earth observation satellites? *Journal of Computational and Graphical Statistics*, 1999. Special Section: Massive Datasets.

Object Management Group. *Common Object Request Broker: Architecture and Specification, Version 2.0*. Framingham, MA, 1996. URL http://www.omg.org.

Duncan Temple Lang. R embedded within the Postgres database server, 2000. URL http://www.omegahat.org/RSPostgres.

Luke Tierney. Notes on references, external objects, or mutable state for R, 2002. URL http://www.stat.uiowa.edu/~luke/R/references.html.

Affiliation

Fei Chen, Brian D. Ripley
Department of Statistics
University of Oxford
1 South Parks Road
Oxford, OX1 3TG
United Kingdom