R News; the Newsletter of the R Project
Vol.3, No.1 June 2003
© R Foundation for Statistical Computing
http://cran.r-project.org/

Computational Gains Using RPVM on a Beowulf Cluster

by Brett Carson, Robert Murison and Ian A. Mason.

Introduction

The Beowulf cluster Becker et al. (1995); Scyld Computing Corporation (1998) is a recent advance in computing technology that harnesses the power of a network of desktop computers using communication software such as PVM Geist et al. (1994) and MPI Message Passing Interface Forum (1997). Whilst the potential of a computing cluster is obvious, expertise in programming is still developing in the statistical community.

Recent articles in R-news Li and Rossini (2001) and Yu (2002) entice statistical programmers to consider whether their solutions could be effectively calculated in parallel. Another R package, SNOW Tierney (2002); Rossini et al. (2003) aims to skillfully provide a wrapper interface to these packages, independent of the underlying cluster communication method used in parallel computing. This article concentrates on RPVM and wishes to build upon the contribution of Li and Rossini (2001) by taking an example with obvious orthogonal components and detailing the R code necessary to allocate the computations to each node of the cluster. The statistical technique used to motivate our RPVM application is the gene-shaving algorithm Hastie et al. (2000b,a) for which S-PLUS code has been written by Do and Wen (2002) to perform the calculations serially.

The first section is a brief description of the Beowulf cluster used to run the R programs discussed in this paper. This is followed by an explanation of the gene-shaving algorithm, identifying the opportunities for parallel computing of bootstrap estimates of the "strength" of a cluster and the rendering of each matrix row orthogonal to the "eigen-gene". The code for spawning child processes is then explained comprehensively and the conclusion compares the speed of RPVM on the Beowulf to serial computing.

A parallel computing environment

The Beowulf cluster Becker et al. (1995); Scyld Computing Corporation (1998) used to perform the parallel computation outlined in the following sections is a sixteen node homogeneous cluster. Each node consists of an AMD XP1800+ with 1.5Gb of memory, running Redhat Linux 7.3. The nodes are connected by 100Mbps Ethernet, and are accessible via the dual processor AMD XP1800+ front-end. Each node has installed versions of R, PVM Geist et al. (1994) and R's PVM wrapper, RPVM Li and Rossini (2001).

Gene-shaving

The gene-shaving algorithm Hastie et al. (2000b,a) is one of the many DNA micro-array gene clustering techniques to be developed in recent times. The aim is to find clusters with small variance between genes, and large variance between samples Hastie et al. (2000b). The algorithm consists of a number of steps:

1. The process starts with the matrix $X_{N,p}$ which represents expressions of genes in the micro array. Each row of X is data from a gene, the columns contain samples. N is usually of size 10^3 and p usually no greater than 10^2.

2. The leading principal component of X is calculated and termed the "eigen-gene".

3. The correlation of each row with the eigen-gene is calculated and the rows yielding the lowest 10% of R^2 are shaved. This process is repeated to produce successive reduced matrices $X_1^*, \ldots X_N^*$. There are $[0.9N]$ genes in X_1^*, $[0.81N]$ in X_2^* and 1 in X_N^*.

4. The optimal cluster size is selected from $\mathbf{X}^*$. The selection process requires that we estimate the distribution of R^2 by bootstrap and calculate $G_k = R_k^2 - R*_k^2$ where $R*_k^2$ is the mean R^2 from B bootstrap samples of X_k. The statistic G_k is termed the gap statistic and the optimal cluster is that which has the largest gap statistic.

5. Once a cluster has been determined, the process repeats searching for more clusters, after removing the information of previously determined clusters. This is done by "sweeping" the mean gene of the optimal cluster, $\bar{x}_{S_{\hat{k}}}$, from each row of X,

$$newX_{[i,]} = X_{[i,]} \underbrace{(I - \bar{x}_{S_{\hat{k}}}(\bar{x}_{S_{\hat{k}}}^T \bar{x}_{S_{\hat{k}}})^{-1} \bar{x}_{S_{\hat{k}}}^T)}_{IPx}$$

6. The next optimal cluster is found by repeating the process using $newX$.

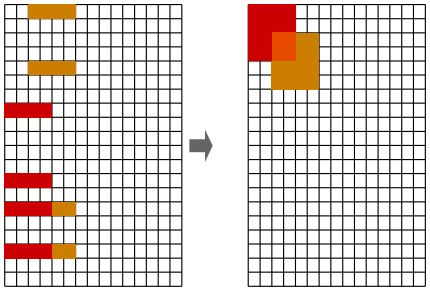

Figure 1: Clustering of genes

The bootstrap and orthogonalization processes are candidates for parallel computing. Figure 1 illustrates the overall gene-shaving process by showing the discovery of clusters of genes.

Parallel implementation of gene-shaving

There are two steps in the shaving process that can be computed in parallel. Firstly, the bootstrapping of the matrix X, and secondly, the orthogonalization of the matrix X with respect to $\bar{x}_{S_k}$. In general, for an algorithm that can be fully parallelized, we could expect a speedup of no more than 16 on a cluster of 16 nodes. However, given that this particular algorithm will not be fully parallelized, we would not expect such a reduction in computation time. In fact, the cluster will not be fully utilised during the overall computation, as most of the nodes will see significant idle time. Amdahl's Law Amdahl (1967) can be used to calculate the estimated speedup of performing gene-shaving in parallel. Amdahl's Law can be represented in mathematical form by:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

where r_s is the proportion of the program computed in serial, r_p is the parallel proportion and n is the number of processors used. In a perfect world, if a particular step computed in parallel were spread over five nodes, the speedup would equal five. This is very rarely the case however, due to speed limiting factors like network load, `PVM` overheads, communication overheads, `CPU` load and the like.

The parallel setup

Both the parallel versions of bootstrapping and orthogonalization consist of a main `R` process (the master), and a number of child processes that the master spawns (the slaves). The master process controls the entire flow of the shaving process. It begins by starting the child processes, which then proceed to wait for the master to send data for processing, and return an answer to the master. There are two varieties of children/slaves:

- Bootstrapping children - each slave task performs one permutation of the bootstrapping process.
- Orthogonalization children - each slave computes a portion of the orthogonalization process.

Figure 2 shows a basic master-slave setup, without any communication between the slaves themselves. The slaves receive some data from the parent, process it, and send the result back to the parent. A more elaborate setup would see the slaves communicating with each other, this is not necessary in this case, as each of the slaves is independent of each other.

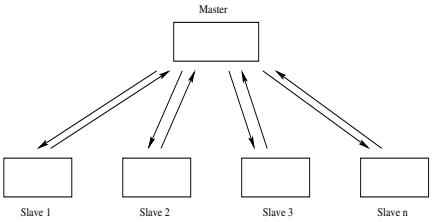

Figure 2: The master-slave paradigm without slave-slave communication.

Bootstrapping

Bootstrapping is used to find $R*^2$ used in the gap statistic outlined in step 4 of the gene-shaving algorithm, to determine whether selected clusters could have arisen by chance. Elements of each row of the matrix are rearranged, which requires N iterations, quite a `CPU` intensive operation, even more so for larger expression matrices. In the serial `R` version, this would be done using `R`'s `apply` function, whereby each row of X has the `sample` function applied to it.

The matrix X is transformed into X^* B times by randomly permuting within rows. For each of the B bootstrap samples (indexed by b), the statistic R_b^2 is derived and the mean of $R_b^2, b = 1, B$ is calculated to represent that R^2 which arises by chance only. In `RPVM`, for each bootstrap sample a task is spawned; for B bootstrap samples we spawn B tasks.

Orthogonalization

The master `R` process distributes a portion of the matrix X to its children. Each portion will be approximately equal in size, and the number of portions will be equal to the number of slaves spawned for orthogonalization. The master also calculates the value of IPx, as defined in step 5 of the gene-shaving algorithm.

The value of IPx is passed to each child. Each child then applies IPx to each row of X that the master has given it.

There are two ways in which X can be distributed. The matrix can be split into approximately even portions, where the number of portions will be equal to the number of slaves. Alternatively, the matrix can be distributed on a row-by-row basis, where each slave processes a row, sends it back, is given a new row to process and so on. The latter method becomes slow and inefficient, as more sending, receiving, and unpacking of buffers is required. Such a method may become useful where the number of rows of the matrix is small and the number of columns is very large.

The RPVM code

Li and Rossini (2001) have explained the initial steps of starting the PVM daemon and spawning RPVM tasks from within an R program, These steps precede the following code. The full source code is available from http://mcs.une.edu.au/~bcarson/RPVM/.

Bootstrapping

Bootstrapping can be performed in parallel by treating each bootstrap sample as a child process. The number of children is the number of bootstrap samples, (B) and the matrix is copied to each child, ($1, ..., B$). Child process i permutes elements within each row of X, calculates $R*^2$ and returns this to the master process. The master averages $R*^2$, $i = 1, B$ and computes the gap statistic.

The usual procedure for sending objects to PVM child processes is to initialize the send buffer, pack the buffer with data and send the buffer. Our function `par.bootstrap` is a function used by the master process and includes `.PVM.initsend()` to initialize the send buffer, `.PVM.pkdblmat(X)` to pack the buffer, and `.PVM.mcast(boot.children,0)` to send the data to all children.

```
par.bootstrap <- function(X){
  brsq <- 0

  tasks <- length(boot.children)

  .PVM.initsend()
  #send a copy of X to everyone
  .PVM.pkdblmat(X)
  .PVM.mcast(boot.children,0)

  #get the R squared values back and
  #take the mean
  for(i in 1:tasks){
    .PVM.recv(boot.children[i],0)
    rsq <- .PVM.upkdouble(1,1)
    brsq <- brsq + (rsq/tasks)
  }
  return(brsq)
}
```

After the master has sent the matrix to the children, it waits for them to return their calculated R^2 values. This is controlled by a simple `for` loop, as the master receives the answer via `.PVM.recv(boot.children[i],0)`, it is unpacked from the receive buffer with:

```
rsq <- .PVM.upkdouble(1,1)
```

The value obtained is used in calculating an overall mean for all R^2 values returned by the children. In the slave tasks, the matrix is unpacked and a bootstrap permutation is performed, an R^2 value calculated and returned to the master. The following code performs these operations of the children tasks:

```
#receive the matrix
.PVM.recv(.PVM.parent(),0)
X <- .PVM.upkdblmat()

Xb <- X
dd <- dim(X)

#perform the permutation
for(i in 1:dd[1]){
  permute <- sample(1:dd[2],replace=F)
  Xb[i,] <- X[i,permute]
}

#send it back to the master
.PVM.initsend()
.PVM.pkdouble(Dfunct(Xb))
.PVM.send(.PVM.parent(),0)
```

`.PVM.upkdblmat()` is used to unpack the matrix. After the bootstrap is performed, the send is initialized with `.PVM.initsend()` (as seen in the master), the result of `Dfunct(Xb)` (R^2 calculation) is packed into the send buffer, and then the answer is sent with `.PVM.send()` to the parent of the slave, which will be the master process.

Orthogonalization

The orthogonalization in the master process is handled by the function `par.orth`. This function takes as arguments the IPx matrix, and the expression matrix X. The first step of this procedure is to broadcast the IPx matrix to all slaves by firstly packing it and then sending it with `.PVM.mcast(orth.children,0)`. The next step is to split the expression matrix into portions and distribute to the slaves. For simplicity, this example divides the matrix by the number of slaves, so row size must be divisible by the number of children. Using even blocks on a homogeneous cluster will work reasonably well, as the load-balancing issues outlined in Rossini et al. (2003) are more relevant to heterogeneous clusters. The final step performed by the master is to receive these portions

back from the slaves and reconstruct the matrix. Order of rows is maintained by sending blocks of rows and receiving them back in the order the children sit in the array of task ids (`orth.children`). The following code is the `par.orth` function used by the master process:

```
par.orth <- function(IPx, X){
  rows <- dim(X)[1]

  tasks <- length(orth.children)
  #intialize the send buffer
  .PVM.initsend()
  #pack the IPx matrix and send it to
  #all children
  .PVM.pkdblmat(IPx)
  .PVM.mcast(orth.children,0)

  #divide the X matrix into blocks and
  #send a block to each child
  n.rows = as.integer(rows/tasks)
  for(i in 1:tasks){
    start <- (n.rows * (i-1))+1
    end <- n.rows * i
    if(end > rows)
      end <- rows
    X.part <- X[start:end,]
    .PVM.pkdblmat(X.part)
    .PVM.send(orth.children[i],start)
  }

  #wait for the blocks to be returned
  #and re-construct the matrix
  for(i in 1:tasks){
    .PVM.recv(orth.children[i],0)
    rZ <- .PVM.upkdblmat()
    if(i==1){
      Z <- rZ
    }
    else{
      Z <- rbind(Z,rZ)
    }
  }
  dimnames(Z) <- dimnames(X)
  Z
}
```

The slaves execute the following code, which is quite similar to the code of the bootstrap slaves. Each slave receives both matrices, then performs the orthogonalization and sends back the result to the master:

```
  #wait for the IPx and X matrices to arrive
  .PVM.recv(.PVM.parent(),0)
  IPx <- .PVM.upkdblmat()
  .PVM.recv(.PVM.parent(),-1)
  X.part <- .PVM.upkdblmat()

  Z <- X.part
  #orthogonalize X
  for(i in 1:dim(X.part)[1]){
    Z[i,] <- X.part[i,] %*% IPx
  }
  #send back the new matrix Z
  .PVM.initsend()
  .PVM.pkdblmat(Z)
  .PVM.send(.PVM.parent(),0)
```

Gene-shaving results

Serial gene-shaving

The results for the serial implementation of gene-shaving are a useful guide to the speedup that can be achieved from parallelizing. The proportion of the total computation time the bootstrapping and orthogonalization use is of interest. If these are of significant proportion, they are worth performing in parallel.

The results presented were obtained from running the serial version on a single cluster node. `RPVM` is not used in the serial version, and the program contains no parallel code at all. The data used are randomly generated matrices of size $N = 1600$, 3200, 6400, and 12800 respectively, with a constant p of size 80, to simulate variable-sized micro-arrays. In each case, the first two clusters are found, and ten permutations used in each bootstrap. Figure 3 is a plot of the overall computation time, the total time of all invocations of the bootstrapping process, and the total time of all invocations of the orthogonalization process.

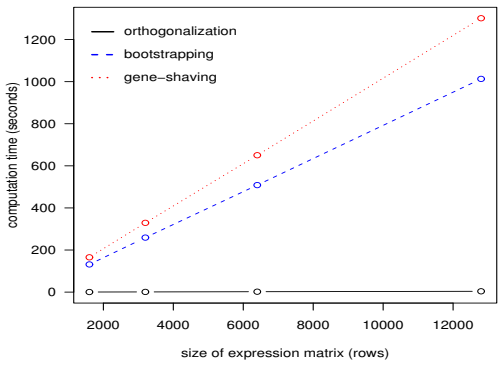

Figure 3: Serial Gene-shaving results

Clearly, the bootstrapping (blue) step is quite time consuming, with the mean proportion being 0.78 of the time taken to perform gene-shaving (red). The orthogonalization (black) equates to no more than 1%, so any speedup through parallelization is not going to have any significant effect on the overall time.

We have two sets of parallel results, one using two cluster nodes, the other using all sixteen nodes of the cluster. In the two node runs we could expect a reduction in the bootstrapping time to be no more than half, i.e a *speedup* of 2. In full cluster runs we could expect a *speedup* of no more than 10, as

we can use ten of the sixteen processors (one processor for each permutation). It is now possible to estimate the expected maximum speedup of parallelization with Amdahl's law, as described earlier for expression matrices of size $N = 12800$ and $p = 80$, given the parallel proportion of 0.78.

Application of Amdahl's law fot the cases ($n = 2, 10$) gives speedups of 1.63 and 3.35 respectively. We have implemented the orthogonalization in parallel, more for demonstration purposes, as it could become useful with very large matrices.

Parallel gene-shaving

The parallel version of gene-shaving only contains parallel code for the bootstrapping and orthogonalization procedures, the rest of the process remains the same as the serial version. Both the runs using two nodes and all sixteen nodes use this same `R` program. In both cases ten tasks are spawned for bootstrapping and sixteen tasks are spawned for orthogonalization. Clearly this is an under-utilisation of the Beowulf cluster in the sixteen node runs, as most of the nodes are going to see quite a lot of idle time, and in fact six of the nodes will not be used to compute the bootstrap, the most `CPU` intensive task of the entire algorithm.

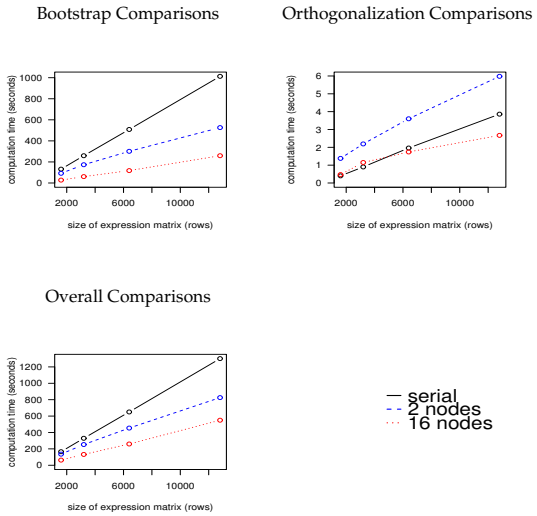

Figure 4: Comparison of computation times for variable sized expression matrices for overall, orthogonalization and bootstrapping under the different environments.

The observed speedup for the two runs of parallel gene-shaving using the $N = 12800$ expression matrix is as follows:

$$S = \frac{T_{serial}}{T_{parallel}}$$

$$S_{2nodes} = \frac{1301.18}{825.34} = 1.6$$

$$S_{16nodes} = \frac{1301.18}{549.59} = 2.4$$

The observed speedup for two nodes is very close to the estimate of 1.63. The results for sixteen nodes are not as impressive, due to greater time spent computing than expected for bootstrapping. Although there is a gain with the orthogonalization here, it is barely noticeable in terms of the whole algorithm. The orthogonalization is only performed once for each cluster found, so it may well become an issue when searching for many clusters on much larger datasets than covered here.

Conclusions

Whilst the gene-shaving algorithm is not completely parallelizable, a large portion of it (almost 80%) of it can be implemented to make use of a Beowulf cluster. This particular problem does not fully utilise the cluster, each node other than the node which runs the master process will see a significant amount of idle time. Regardless, the results presented in this paper show the value and potential of Beowulf clusters in processing large sets of statistical data. A fully parallelizable algorithm which can fully utilise each cluster node will show greater performance over a serial implementation than the particular example we have presented.

Bibliography

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS spring joint computer conference*, Sunnyvale, California. IBM. 22

Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V. (1995). BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION. In *International Conference on Parallel Processing*. 21

Do, K.-A. and Wen, S. (2002). Documentation. Technical report, Department of Biostatistics, MD Anderson Cancer Center. http://odin.mdacc.tmc.edu/ kim/. 21

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing*. MIT Press. 21

Hastie, T., Tibshirani, R., Eisen, M., Alizadeh, A., Levy, R., Staudt, L., Chan, W., Botstein, D., and Brown, P. (2000a). 'Gene shaving' as a method for identifying distinct sets of genes with similar expression patterns. *Genome Biology*. 21

Hastie, T., Tibshirani, R., Eisen, M., Brown, P., Scherf, U., Weinstein, J., Alizadeh, A., Staudt, L., and Botstein, D. (2000b). Gene shaving: a new class of clustering methods for expression arrays. Technical report, Stanford University. 21

Li, M. N. and Rossini, A. (2001). RPVM: Cluster statistical computing in R. *R News*, 1(3):4–7. 21, 23

Message Passing Interface Forum (1997). MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/docs.html. 21

Rossini, A., Tierney, L., and Li, N. (2003). Simple parallel statistical computing in r. Technical Report Working Paper 193, UW Biostatistics Working Paper Series. http://www.bepress.com/uwbiostat/paper193. 21, 23

Scyld Computing Corporation (1998). Beowulf introduction & overview. http://www.beowulf.org/intro.html. 21

Tierney, L. (2002). Simple Network of Workstations for R. Technical report, School of Statistics, University of Minnesota. http://www.stat.uiowa.edu/ luke/R/cluster/cluster.html. 21

Yu, H. (2002). Rmpi: Parallel statistical computing in R. *R News*, 2(2):10–14. 21

Brett Carson, Robert Murison, Ian A. Mason
The University of New England, Australia
bcarson@turing.une.edu.au
rmurison@turing.une.edu.au
iam@turing.une.edu.au

R Help Desk

Getting Help – R's Help Facilities and Manuals

Uwe Ligges

Introduction

This issue of the *R Help Desk* deals with its probably most fundamental topic: Getting help!

Different amounts of knowledge about R and R related experiences require different levels of help. Therefore a description of the available manuals, help functions within R, and mailing lists will be given, as well as an "instruction" how to use these facilities.

A secondary objective of the article is to point out that manuals, help functions, and mailing list archives are commonly much more comprehensive than answers on a mailing list. In other words, there are good reasons to read manuals, use help functions, and search for information in mailing list archives.

Manuals

The manuals described in this Section are shipped with R (in directory '.../doc/manual' of a regular R installation). Nevertheless, when problems occur before R is installed or the available R version is outdated, recent versions of the manuals are also available from CRAN[1]. Let me cite the "R Installation and Administration" manual by the *R Development Core Team* (R Core, for short; 2003b) as a reference for information on installation and updates not only for R itself, but also for contributed packages.

I think it is not possible to use adequately a programming language and environment like R without reading any introductory material . Fundamentals for using R are described in "An Introduction to R" (Venables et al., 2003), and the "R Data Import/Export" manual (R Core, 2003a). Both manuals are the first references for any programming tasks in R.

Frequently asked questions (and corresponding answers) are collected in the "R FAQ" (Hornik, 2003), an important resource not only for beginners. It is a good idea to look into the "R FAQ" when a question arises that cannot be answered by reading the other manuals cited above. Specific FAQs for Macintosh (by Stefano M. Iacus) and Windows (by Brian D. Ripley) are available as well[2].

Users who gained some experiences might want to write their own functions making use of the language in an efficient manner. The manual "R Language Definition" (R Core, 2003c, still labelled as "draft") is a comprehensive reference manual for a couple of important language aspects, e.g.: objects, working with expressions, working with the language (objects), description of the parser, and *debugging*. I recommend this manual to all users who plan to work regularly with R, since it provides really illuminating insights.

For those rather experienced users who plan to create their own packages, writing help pages and optimize their code, "Writing R Extensions" (R Core,

[1] http://CRAN.R-project.org/
[2] http://CRAN.R-project.org/faqs.html