(restricted) SARIMA models. To motivate this, consider equation 2—the error-correction form of exponential smoothing. By plugging it into the forecasting equation 1, we almost directly obtain:

$$\begin{aligned}(1-L)\hat{Y}_t &= \alpha e_{t-1} \\ &= (1-\theta L)e_t\end{aligned}$$

($\theta = 1-\alpha$), which is (in terms of estimates) the standard form of an ARIMA(0,1,1)-process. Finally, we note that we actually face *state-space* models: given process Y_t, we try to estimate the underlying processes a_t, b_t and s_t which cannot be observed directly. But this is the story of structural time series and the function `structTS`, told in an another article by Brian D. Ripley ...

Bibliography

Holt, C. (1957). Forecasting seasonals and trends by exponentially weighted moving averages. *ONR Research Memorandum, Carnegie Institute 52*. 7

Hyndman, R., Koehler, A., Snyder, R., & Grose, S. (2001). A state space framework for automatic forecasting using exponential smoothing methods. *Journal of Forecasting*. To appear. 9

Winters, P. (1960). Forecasting sales by exponentially weighted moving averages. *Management Science*, **6**, 324–342. 7

David Meyer
Technische Universität Wien, Austria
David.Meyer@ci.tuwien.ac.at

Rmpi: Parallel Statistical Computing in R

by Hao Yu

Rmpi is an R interface (wrapper) to the Message-Passing Interface (MPI). MPI is a standard application interface (API) governed by the MPI forum (http://www.mpi-forum.org) for running parallel applications. Applicable computing environments range from dedicated Beowulf PC clusters to parallel supercomputers such as IBM's SP2. Performance and portability are the two main strengths of MPI. In this article, we demonstrate how the MPI API is implemented in **Rmpi**. In particular, we show how interactive R slaves are spawned and how we use them to do sophisticated MPI parallel programming beyond the "embarrassingly parallel".

Introduction

Put simply, parallel statistical computing means dividing a job into many small parts which will be executed simultaneously over a cluster of computers linked together in a network (LAN or WAN). Communications among the components is a crucial aspect of parallel computing. The message-passing model is highly suitable for such a task. There are two primary open-source message-passing models available: MPI and PVM (Parallel Virtual Machine, Geist et al., 1994). For PVM's implementation **rpvm** in R, see Li and Rossini, 2001.

Whether to use MPI or PVM largely depends on the underlying hardware structure and one's personal preference. Both models provide a parallel programming interface. The main difference is in how buffers are managed or used for sending and receiving data. Typically, in PVM, users are responsible for packing data into a system designated buffer on the sender side and unpacking from a system buffer on the receiver side. This reduces the user's responsibility for managing the buffer; on the other hand, this may create problems of running out of buffer space if large amounts of data are transferred. PVM can also lead to performance draining copies. MPI, by contrast, requires no buffering of data within the system. This allows MPI to be implemented on the widest range of platforms, with great efficiency. Perhaps Luke Tierney's package **snow** (Simple Network of Workstations, http://www.stat.umn.edu/~luke/R/cluster) will ultimately provide a unified interface to both **Rmpi** and **rpvm** so users can freely choose either one to implement parallel computing.

There are 128 routines in the MPI-1 standard and 287 functions in the combined MPI (MPI-1 and MPI-2) standard. MPI is rich in functionality; it is also capable of handling the diversity and complexity of today's high performance computers. Likely the environment with highest potential for MPI use is the Beowulf cluster, a high-performance massively parallel computer built primarily out of commodity hardware components.

One of the goals of **Rmpi** is to provide an extensive MPI API in the R environment. Using R as an interface, the user can either spawn C(++) or Fortran programs as children or join other MPI programs. Another goal is to provide an interactive R master and slave environment so MPI programming can be done completely in R. This was achieved by implementing a set of MPI API extensions specifically designed for the R or R slave environments.

Installation

Although MPI is a standard, its form does not necessitate implementation of all defined functions. There are at least ten freely available implementations of MPI, the most notable of which are MPICH (http://www.mcs.anl.gov/mpi/mpich) and LAM-MPI (http://www.lam-mpi.org). **Rmpi** uses LAM-MPI primarily because its implementation of the MPI API is more comprehensive than other implementations. However, **Rmpi** tries to use as few LAM specific functions as possible to avoid portability problems.

Before attempting installing LAM and **Rmpi**, a Beowulf cluster should be set up properly. See http://www.beowulf.org for details.

Installing LAM

The source code or pre-compiled Linux RPMs can be downloaded from the LAM web site. When attempting to install LAM at the system level, one should ensure that there is no other MPI, such as MPICH, already installed. Two MPIs cannot coexist at the system level. Once LAM is installed, edit the file 'lam-bhost.def' either in the '/etc/lam' system directory or 'etc' in the user's home directory to add all hosts in a cluster. LAM uses `rsh` by default for remote execution. This can be changed to `ssh`. Check LAM and SSH documents for setting ssh and public key authentication to avoid passwords.

Use `lamboot` to boot LAM environment and run `lamexec C hostname` to see if all hosts respond. LAM provides a set of tools for host management and MPI program monitoring (`mpitask`).

Installing Rmpi

For LAM installed in '/usr' or '/usr/local' or for the Debian system with installed packages **lam3**, **lam3-dev**, and **lam-runtime**, just use

```
R CMD INSTALL Rmpi_version.tar.gz
```

For LAM installed in another location, please use

```
R CMD INSTALL Rmpi_version.tar.gz
  --configure-args=--with-mpi=/mpipath
```

Rmpi relies on Luke Tierney's package **serialize** for sending or receiving arbitrary R objects across the network and it must be installed. If you want to use any random number distributions in parallel environment, please install Michael Li's package **rsprng**: a wrapper to SPRNG (Scalable Parallel Random Number Generators).

A sample Rmpi session

```
{karl:10} lamboot
{karl:11} R
> library(Rmpi)
Rmpi version: 0.4-4
Rmpi is an interface (wrapper) to MPI APIs
with interactive R slave functionalities.
See 'library (help=Rmpi)' for details.
Loading required package: serialize
> mpi.spawn.Rslaves(nslaves=3)
3 slaves are spawned successfully. 0 failed.
master (rank 0,comm 1) of size 4 is running
    on: karl
slave1 (rank 1,comm 1) of size 4 is running
    on: karl
slave2 (rank 2,comm 1) of size 4 is running
    on: karl
slave3 (rank 3,comm 1) of size 4 is running
    on: karl4
> mpi.remote.exec(mean(rnorm(1000)))
           X1          X2          X3
1 -0.04475399 -0.04475399 -0.04475399
> mpi.bcast.cmd(mpi.init.sprng())
> mpi.init.sprng()
Loading required package: rsprng
> mpi.remote.exec(mean(rnorm(1000)))
            X1            X2          X3
1 -0.001203990 -0.0002667920 -0.04333435
> mpi.bcast.cmd(free.sprng())
> mpi.close.Rslaves()
[1] 1
> free.sprng()
> mpi.exit()
[1] "Detaching Rmpi. Rmpi cannot be used
    unless relaunching R."
> q()
{karl:12} lamhalt
```

MPI implementation in R

MPI uses a number of objects for message-passing. The most important one is a *communicator*. There are two types of communicators: *intracommunicator* and *intercommunicator*. A *communicator* (*comm*) usually refers to an *intracommunicator* which is associated with a group of members (CPU nodes). Most point-to-point and collective operations among the members must go through it. An *intercommunicator* is associated with two groups instead of members.

Rmpi defines several pointers in system memory to represent commonly used MPI objects. When loading **Rmpi**, an array of size 10 is allocated for *comm* objects, and arrays of size 1 are allocated for *status* and *info* objects. These objects are addressed using the R argument assignment. For example, `comm = 0` represents *comm* object 0 which by default is assigned to MPI_COMM_WORLD. MPI datatypes integer, double, and character are represented by `type=1`, `type=2`, and `type=3` respectively. They match R's integer, double, and character datatypes. Other types require serialization.

A general R object can be serialized to characters (`type=3`) before sending and unserialized after receiving. On a heterogeneous environment, MPI takes

care of any necessary character conversion provided characters are represented by the same number of bytes on both sending and receiving systems.

Rmpi retains the original MPI C interface. A notable exception is the omission of message length because of R's way of determining length.

Regarding receive buffer preparation, one can use `integer(n)` (`double(n)`) for an integer buffer (a double buffer) of size `n`. However, R does not have a function for creating a character buffer (`character(1)` only creates an empty string). The function `string(n)` is supplied to create a character vector with one length `n` element. For example,

```
> string(2)
[1] "  "
```

`string` can be used as a receiver buffer for either a character vector of length 1 or a binary character vector generated by serialization.

In `.First.lib`, the LAM/MPI runtime environment is checked by a LAM command `lamnodes`. If it is not up, `lamboot` will be launched to boot a LAM session. After **Rmpi** is loaded, R becomes an MPI master with one member only (i.e., itself). Then it can use `mpi.comm.spawn` to spawn children. During spawning, an *intercommunicator* (default to *comm* 2) is created. The master (group 1) and children (group 2) should use intercomm merger so they will be in the same group for point-to-point or collective operations.

Interactive R slaves

In this section, we give details on how to spawn R slaves and communicate with them. The main function is `mpi.spawn.Rslaves`. This spawns a set of R slaves by using a shell program 'Rslaves.sh' in the Rmpi installation directory. The slaves will run on nodes specified by either LAM or a user. 'Rslave.sh' usually makes R run in BATCH mode so input (Rscript) and output (log) are required. Those log files are uniquely named after their host names, master process id, and master comm number. They can be very useful when debugging.

The default Rscript, 'slavedaemon.R' in the Rmpi installation directory, performs the necessary steps (intercomm merger) to establish communication with the master and runs in a while loop (waiting for an instruction from the master). When slaves are spawned, they form their own group accessed by *comm* 0. After intercomm merger, slaves use the default *comm* `.comm` (=1) to communicate with the master while the master uses the default *comm* 1 to communicate with the slaves. If necessary, the master can spawn additional sets of R slaves by using *comm* 3, 4, etc. (slaves still use `.comm=1`).

To determine the maximum number of slaves to be spawned, **Rmpi** provides a function `mpi.universe.size` to show the total nodes (CPUs) available in a LAM session. The master should not participate in numerical computation since the number of master and slaves may then exceed the number of nodes available.

In a heterogeneous environment, the user may wish to spawn slaves to specific nodes to achieve uniform CPU speed. For example,

```
> lamhosts()
 karl  karl karl4 karl4 karl5 karl5
    0     1     2     3     4     5
> mpi.spawn.Rslaves(hosts=c(3, 4), comm=3)
2 slaves are spawned successfully. 0 failed.
master (rank 0,comm 3) of size 3 is running
    on: karl
slave1 (rank 1,comm 1) of size 3 is running
    on: karl4
slave2 (rank 2,comm 1) of size 3 is running
    on: karl5
```

Once R slaves are spawned (assuming 3 slaves on *comm* 1 and 2 slaves on *comm* 3), the master can use `mpi.bcast.cmd` to send a command to be executed by all slaves. For example,

```
> mpi.bcast.cmd(print(date()), comm=1)
```

will let all slaves (associated with *comm* 1) execute the command `print(date())` and the results can be viewed by `tail.slave.log(comm=1)`. The command

```
> mpi.bcast.cmd(print(.Last.value), comm=1)
```

tells the slaves to put their last value into the log. If the executed results should be returned to master, one can use

```
> mpi.remote.exec(date(), comm=1)
```

We can think of `mpi.remote.exec` as a parallel `apply` function. It executes "embarrassingly parallel" computations (i.e., those where no node depends on any other). If the master writes a function intended to be executed by slaves, this function must be transferred to them first. One can use `mpi.bcast.Robj2slave` for that purpose. If each slave intends to do a different job, `mpi.comm.rank` can be used to identify itself:

```
> mpi.remote.exec(mpi.comm.rank(.comm),
    comm=1)
  X1 X2 X3
1  1  2  3
> mpi.remote.exec(1:mpi.comm.rank(.comm),
    comm=3)
$slave1
[1] 1
$slave2
[1] 1 2
```

Often the master will let all slaves execute a function while the argument values are on the master. Rather than relying on additional MPI call(s), one can pass the original arguments as in the following example.

```
> x <- 1:10
> mpi.remote.exec(mean, x, comm=1)
   X1  X2  X3
1 5.5 5.5 5.5
```

Here, the slaves execute `mean(x)` with x replaced by 1:10. Note that `mpi.remote.exec` allows only a small amount of data to be passed in this way.

Example

Rmpi comes with several demos. The following two functions are similar to the demo script 'slave2PI.R'.

```
slave2 <- function(n) {
  request <-1; job <-2
  anytag <- mpi.any.tag(); mypi <- 0
  while (1) {
    ## send master a request
    mpi.send(integer(1), type=1, dest=0,
      tag=request,comm=.comm)
    jobrange<-mpi.recv(integer(2), type=1,
      source=0, tag=anytag, comm=.comm)
    tag <- mpi.get.sourcetag()[2]
    if (tag==job) #do the computation
      mypi <- 4*sum(1/(1+((seq(jobrange[1],
        jobrange[2])-.5)/n)^2))/n + mypi
    else break #tag=0 means stop
  }
  mpi.reduce(mypi, comm=.comm)
}
master2PI <- function(n, maxjoblen, comm=1) {
  tsize <- mpi.comm.size(comm)
  if (tsize < 2)
    stop("Need at least 1 slave")
  ## send the function slave2 to all slaves
  mpi.bcast.Robj2slave(slave2, comm=comm)
  #let slave run the function slave2
  mpi.remote.exec(slave2, n=n, comm=comm,
        ret=FALSE)
  count <- 0; request <- 1; job <- 2
  anysrc <- mpi.any.source()
  while (count < n) {
    mpi.recv(integer(1), type=1,
      source=anysrc, tag=request, comm=comm)
    src <- mpi.get.sourcetag()[1]
    jobrange <- c(count+1,
            min(count+maxjoblen, n))
    mpi.send(as.integer(jobrange), type=1,
      dest=src, tag=job, comm=comm)
    count <- count+maxjoblen
  }
  ## tell slaves to stop with tag=0
  for (i in 1:(tsize-1)) {
    mpi.recv(integer(1), type=1,
      source=anysrc, tag=request, comm=comm)
    src <- mpi.get.sourcetag()[1]
    mpi.send(integer(1), type=1,
      dest=src, tag=0, comm=comm)
  }
  mpi.reduce(0, comm=comm)
}
```

The function slave2 is to be executed by all slaves and master2PI is to be run on the master. The computation is simply a numerical integration of $\int_0^1 1/(1+x^2)\,dx$. qA load balancing approach is used, namely, the computation is divided into more small jobs than there are slaves so that whenever a slave finishes a job, it can request a new one and continue. This approach is particularly useful in a heterogeneous environment where CPU's speeds differ.

Notice that the wild card `mpi.any.source` is used by the master in `mpi.recv` for first-come-first-served (FCFS) type of queue service. During computation, slaves keep their computed results until the last step `mpi.reduce` for global reduction. Since the master is a member, it must do the same by adding 0 to the final number. For 10 small jobs computed by 3 slaves, one can

```
> master2PI(10000, 1000, comm=1) - pi
[1] 8.333334e-10
```

By replacing the slaves' computation part, one can easily modify the above codes for other similar types of parallel computation.

On clusters with other MPIs

Several parallel computer vendors implement their MPI based on MPICH without spawning functionality. Can **Rmpi** still be used with interactive R slaves capability? This will not be an issue if one installs LAM at the user level. However, this practice is often not allowed for two reasons. First, vendors already optimize MPIs for their hardware to maximize network performance; LAM may not take advantage of this. Second, a job scheduler is often used to dispatch parallel jobs across nodes to achieve system load balancing. A user-dispatched parallel job will disrupt the load balance.

To run **Rmpi** in such clusters, several steps must be taken. The detailed steps are given in the 'README' file. Here we sketch these steps.

1. Modify and install **Rmpi** with the system supplied MPI;
2. Copy and rename the file 'Rprofile' in the Rmpi installation directory to the user's root directory as '.Rprofile';
3. Run `mpirun -np 5 R -save -q` to launch 1 master and 4 slaves with default *comm* 1.

On some clusters `mpirun` may dispatch the master to a remote node which effectively disables R interactive mode. **Rmpi** will still work in pseudo-interactive mode with command line editing disabled. One can run R in `R CMD BATCH` mode as `mpirun -np 5 R CMD BATCH` *Rin Rout*. Notice that only the master executes the Rscript *Rin*, exactly the same as in interactive mode.

Discussion

MPI has many routines. It is not currently feasible to implement them all in **Rmpi**. Some of the routines are not necessary. For example, many of the

data management routines are not needed, since R has its own sophisticated data subsetting tools.

Virtual topology is a standard feature of MPI. Topologies provide a high-level method for managing CPU groups without dealing with them directly. The *Cartesian* topology implementation will be the target of future version of **Rmpi**.

MPI profiling is an interesting area that may enhance MPI programming in R. It remains to be seen if the MPE (Multi-Processing Environment) library can be implemented in **Rmpi** or whether it will best be implemented as a separate package.

Other exotic advanced features of MPI under consideration are Parallel I/O and Remote Memory Access (RMA) for one-sided communication.

With parallel programming, debugging remains a challenging research area. Deadlock (i.e., a situation arising when a failed task leaves a thread waiting) and race conditions (i.e., bugs caused by failing to account for dependence on the relative timing of events) are always issues regardless of whether one is working at a low level (C(++) or Fortran) or at a high level (R). The standard MPI references can provide help.

Acknowledgements

Rmpi is primarily implemented on the SASWulf Beowulf cluster funded by NSERC equipment grants. Support from NSERC is gratefully acknowledged.

Thanks to my colleagues Drs. John Braun and Duncan Murdoch for proofreading this article. Their effects have made this article more readable.

Bibliography

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

W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI-The Complete Reference: Volume 2, MPI-2 Extensions*. The MIT Press, Massachusetts, 1998.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Massachusetts, 1999a.

W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Massachusetts, 1999b.

Michael Na Li and A.J. Rossini. RPVM: Cluster statistical computing in R. *R News*, 1(3):4–7, September 2001. URL http://CRAN.R-project.org/doc/Rnews/.

M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference: Volume 1, The MPI Core*. The MIT Press, Massachusetts, 1998.

Hao Yu
University of Western Ontario
hyu@stats.uwo.ca

The grid Graphics Package

by Paul Murrell

Introduction

The **grid** package provides an alternative set of graphics functions within R. This article focuses on **grid** as a drawing tool. For this purpose, **grid** provides two main services:

1. the production of low-level to medium-level graphical components such as lines, rectangles, data symbols, and axes.

2. sophisticated support for arranging graphical components.

The features of **grid** are demonstrated via several examples including code. Not all of the details of the code are explained in the text so a close consideration of the output and the code that produced it, plus reference to the on-line help for specific **grid** functions, may be required to gain a complete understanding.

The functions in **grid** do not provide complete high-level graphical components such as scatterplots or barplots. Instead, **grid** is designed to make it very easy to build such things from their basic components. This has three main aims:

1. The removal of some of the inconvenient constraints imposed by R's default graphical functions (e.g., the fact that you cannot draw anything other than text relative to the coordinate system of the margins of a plot).

2. The development of functions to produce high-level graphical components which would not be very easy to produce using R's default graphical functions (e.g., the **lattice** add-on package for R, which is described in a companion article in this issue).

3. The rapid development of novel graphical displays.