to have a dyld/Mach-O Carbon version, truly native to OS X. The Quartz driver also brings us closer to a Cocoa version of R, which could be implemented initially as a Cocoa shell around the Darwin version of R.

Much will depend on the reception of OS X, and on how many Mac users will switch from 9.x to X. If your hardware supports OS X, I think switching is a no-brainer, especially if you program, develop, or compute. As I have indicated above, the possibilities are endless.

Jan de Leeuw
University of California at Los Angeles
deleeuw@stat.ucla.edu

RPVM: Cluster Statistical Computing in R

by Michael Na Li and A.J. Rossini

rpvm is a wrapper for the *Parallel Virtual Machine* (PVM) API. PVM (Geist et al., 1994) is one of the original APIs for extending an application over a set of processors in a parallel computer or over machines in a local area cluster. We discuss the PVM API, how it is implemented in R, and provide examples for its use. **rpvm** provides a quick means for prototyping parallel statistical applications as well as for providing a front-end for data analysis from legacy PVM applications.

Introduction

PVM was developed at Oak Ridge National Laboratories and the University of Tennessee starting in 1989. It is a *de facto* standard for distributed computing designed especially for heterogeneous networks of computers. The notion of "virtual machine" makes the network appear logically to the user as a single large parallel computer. It provides a mechanism for specifying the allocation of tasks to specific processors or machines, both at the start of the program as well as dynamically during runtime. There are routines for the two main types of intertask communication: point-to-point communication between tasks (including broadcasting) and collective communication within a group of tasks.

The primary message passing library competitor to PVM is MPI (*Message Passing Interface*). The biggest advantage of PVM over MPI is its flexibility (Geist et al., 1996). PVM can be run on an existing network consisting of different platforms (almost all platforms are supported, including Microsoft Windows 98/NT/2000 systems). Tasks can be dynamically spawned, which is not supported in MPI-1 upon which most MPI implementations are based. Hosts can be dynamically added or deleted from the virtual machine, providing fault tolerance. There are also a visualization tool, `xpvm`, and numerous debugging systems. MPI has advantages of speed as well as being an actual standard. However, for prototyping and research, it isn't clear that either of these are critical features.

PVM has been successfully applied to many applications, such as molecular dynamics, semiconductor device simulation, linear algebra (ScaLAPACK, NAG PVM library), etc. It also has great potential in statistical computing, including optimization (expensive or large number of function evaluations; likelihood computations), simulations (resampling, including bootstrap, jackknife, and MCMC algorithms; integration), enumeration (permutation and network algorithms), solution of systems of equations (linear, PDE, finite-element, CFD).

This article presents a new R package, **rpvm**, that provides an interface to PVM from one of the most powerful and flexible statistical programming environments. With **rpvm**, the R user can invoke either executable programs written in compiled language such as C, C++ or FORTRAN as child tasks or spawn separate R processes. It is also possible to spawn R processes from other programs such as Python, C, FORTRAN, or C++. Therefore **rpvm** is ideal for prototyping parallel statistical algorithms and for splitting up large memory problems. Using **rpvm**, statisticians will be able to prototype difficult statistical computations easily in parallel. The rest of the article which follows looks at installation, features, a programming example, and concludes with issues for on-going development.

Installation

PVM source code can be downloaded from `http://www.netlib.org/pvm3/pvm3.4.3.tgz`. Binary distributions exist for many Linux distributions (see individual distributions) as well as for Microsoft Windows NT/2000/XP. However, the Windows implementation of **rpvm** is untried (it is possible to communicate with C or FORTRAN processes running under Microsoft Windows). The following procedures refer to UNIX-like environments.

Installing PVM

Compiling the source code: Installation from the source is straightforward. After untarring the source package, set the environment variable PVM_ROOT to where pvm resides, for example '$HOME/pvm3' or '/usr/local/pvm3'. Then type 'make' under the '$PVM_ROOT' directory. The libraries and executables are installed in '$PVM_ROOT/lib/$PVM_ARCH', where PVM_ARCH is the host architecture name, e.g., 'LINUX' or 'SUN4SOL2'. This way one can build PVM for different architectures under the same source tree.

PVM comes with plenty of examples, see the PVM documentation on how to build and run these.

Setting up PVM environment: Before running PVM, some environment variables need to be set. For example, if you use a C shell, put the following in the '$HOME/.cshrc' file of each host,

```
setenv PVM_ROOT $HOME/pvm3
setenv PVM_ARCH '$PVM_ROOT/lib/pvmgetarch'
set path = ( $path $PVM_ROOT/lib \
             $PVM_ROOT/lib/$PVM_ARCH \
             $PVM_ROOT/bin/$PVM_ARCH )
```

PVM uses rsh by default to initialize communication between hosts. To use ssh (Secure Shell) instead, which is necessary for many networks, define

```
setenv PVM_RSH 'which ssh'
```

You can use public key authentication to avoid typing passwords; see the SSH documentation on how to do this.

Setting up RPVM

rpvm uses a shell script '$R_LIBS/rpvm/slaveR.sh' to start a slave R process. After installing **rpvm**, copy this file to '$PVM_ROOT/bin/$PVM_ARCH' so it can be found by the pvm daemon. The path to the slave R script and the slave output file can either be specified through environment variables RSLAVEDIR, RSLAVEOUT or by passing corresponding arguments to the spawning function. The first method can be used when different paths are needed for different host. When the hosts use a shared file system, the second method provides more flexibility. If neither are set, their default values '$R_LIBS/rpvm' and '$TMPDIR' are used.

A sample RPVM session

Below is a sample **rpvm** session. We start the virtual machine by using a host file, '$HOME/.xpvm_hosts',

```
> library(rpvm)
> hostfile <-
+   file.path(Sys.getenv("HOME"), ".xpvm_hosts")
> .PVM.start.pvmd (hostfile)
libpvm [t40001]: pvm_addhosts():
  Already in progress
libpvm [t40001]: pvm_addhosts():
  Already in progress
[1] 0
> .PVM.config()
There are 2 hosts and 2 architectures.
  host.id    name     arch speed
1  262144  abacus    LINUX  1000
2  524288   atlas SUN4SOL2  1000
```

A host file is a simple text file specifying the host names of the computers to be added to the virtual machine. A simple example is shown below.

```
* ep=$HOME/bin/$PVM_ARCH
atlas
abacus
```

where * defines a global option for all hosts. ep=*option* tells the execution path in which we want pvm daemon to look for executables. For more information, please refer to the PVM documentation.

In directory '$R_LIBS/rpvm/demo', there is a test script 'pvm_test.R' which spawns itself as a slave and receives some messages from it.

```
> source(file.path(Sys.getenv("R_LIBS"),
   "rpvm", "demo", "pvm_test.R"))
## Spawning  1 children
### Spawned  1 Task, waiting for data
Message received from  262165
Hello World! from abacus
Some integers  10 7 13
Some doubles  11.7633 11.30661 10.45883
And a matrix
            [,1]        [,2]       [,3]
[1,] -0.76689970 -1.08892973 -0.1855262
[2,] -0.08824007  0.26769811 -1.1625034
[3,]  1.27764749  0.05790402 -1.0725616
Even a factor!
 [1] s t a t i s t i c s
Levels:  a c i s t
```

If this example fails, check to make sure that '$R_LIBS/rpvm/slaveR.sh' is in the executable search path of the pvm daemon and pvm is running.

Features

rpvm provides access to the low-level PVM API as well as to higher-level functions for passing complex R data types such as matrices and factors. Future development will work at extensions to lists and data frames as well as eventually to functions and closures.

Specifically, APIs are provided for the following tasks:

- Virtual Machine Control: to start the virtual machine, add and delete hosts, query the configuration of VM and nodes status, shut down the VM.

- Task Control: to enter and exit from pvm, to start and stop children tasks, query task running status, etc.

- Message Passing: to prepare and send message buffers, to receive message with or without blocking or with timeout, to pack and unpack data, etc.

- Miscellaneous functions to set and get pvm global options, etc.

The implementation currently lacks the functions for Task Grouping, which is planned for the next release.

rpvm also aims in the long run to provide some general purpose functionality for some "naturally" parallel problems (known as "embarrassingly" parallel to computer scientists), such as parallel "apply" (function `PVM.rapply` in the associated script 'slapply.R' being the first attempt) as well as common tasks such as simple Monte Carlo algorithms for bootstrapping.

Using RPVM

Strategies for parallel programming

One common approach to parallel program design (Buyya, 1999) is a master-slave paradigm where one of the tasks is designated the master task and the rest are slave tasks. In general, the master task is responsible for spawning the slave tasks, dividing and sending workload, collecting and combining results from the slaves. The slave tasks only participate in the computation being assigned. Depending on the algorithm, the slaves may or may not communicate among themselves. For PVM, the process is summarized as Master tasks:

- Register with PVM daemon.

- Spawn slaves.

- Send Data.

- Collect and combine results.

- Return and quit.

and Slave tasks:

- Register with PVM daemon.

- Locate parent.

- Receive Data.

- Compute.

- Send results

- Quit.

Alternatively, instead of a star-like topology, one might consider a tree-like process where each task decides if it should split sub-tasks (and later join) or compute and return. Each task is the master to its children and a slave to its parent. This strategy is natural for "divide and conquer" algorithms and a variant of the master-slave paradigm. This might look like:

- Register with PVM daemon

- Determine if I'm the parent or a spawned process.

- Receive data if spawned (already have data if parent).

- Determine if I compute, or if I let slaves compute.

- If slaves compute:
  - Spawn slaves.
  - Send data to slaves.
  - Receive data from slaves.

- Compute.

- If spawned, send results to parent.

- Quit.

This may involve more message passing overhead but may be more efficient for some problems or network architectures and topologies.

Example

`.PVM.rapply` implements a preliminary version of parallel apply function. It divides a matrix up by rows, sends the function to apply and the submatrices to slave tasks and collects the results at the end. It is assumed that the slave script knows how to evaluate the function and returns a scalar for each row.

```
PVM.rapply <-
function(X, FUN = mean, NTASK = 1) {
    ## arbitrary integers tag message intent
    WORKTAG  <- 22
    RESULTAG <- 33
    end   <- nrow(X)
    chunk <- end %/% NTASK + 1
    start <- 1
    ## Register process with pvm daemon
    mytid <- .PVM.mytid()
    ## Spawn R slave tasks
    children <- .PVM.spawnR(ntask = NTASK,
                            slave = "slapply")
    ## One might check if spawning successful,
    ## i.e. entries of children >= 0 ...
    ## If OK then deliver jobs
    for(id in 1:length(children)) {
        ## for each child
        ## initialize message buffer for sending
        .PVM.initsend()
        ## Divide the work evenly (simple-minded)
        range <- c(start,
                   ifelse((start+chunk-1) > end,
                   end,start+chunk-1))
        ## Take a submatrix
        work <-
            X[(range[1]):(range[2]),,drop=FALSE]
        start <- start + chunk
        ## Pack function name as a string
        .PVM.pkstr(deparse(substitute(FUN)))
        ## Id identifies the order of the job
        .PVM.pkint(id)
        ## Pack submatrix
        .PVM.pkdblmat(work)
        ## Send work
        .PVM.send(children[id], WORKTAG)
    }
    ## Receive any outstanding result
    ## (vector of doubles) from each child
    partial.results <- list()
    for(child in children) {
       ## Get message of type result from any
       ## child.
       .PVM.recv(-1, RESULTAG)
       order  <- .PVM.upkint()
       ## unpack result and restore the order
       partial.results[[order]] <-
           .PVM.upkdblvec()
    }
    ##  unregister from pvm
    .PVM.exit()
    return(unlist(partial.results))
}
```

The corresponding slave script 'slapply.R' is

```
WORKTAG <- 22;  RESULTAG <- 33
## Get parent task id and register
myparent <- .PVM.parent()
## Receive work from parent (a matrix)
buf <- .PVM.recv(myparent, WORKTAG)
## Get function to apply
func <- .PVM.upkstr()
## Unpack data (order, partial.work)
order <- .PVM.upkint()
partial.work <- .PVM.upkdblmat()
## actual computation, using apply
partial.result <- apply(partial.work,1,func)
## initialize send buffer
.PVM.initsend()
## pack order and partial.result
.PVM.pkint(order)
.PVM.pkdblvec(partial.result)
## send it back
.PVM.send(myparent, RESULTAG)
## unregister and exit from PVM
.PVM.exit()
```

An even division of jobs may be far from an optimal strategy, which depends on the problem and in this case, on the network architecture. For example, if some nodes in the cluster are significantly faster than others, one may want send more work to them, but this might be counterbalanced by network distance. Computational overhead (more computation in dividing jobs, network activity due to message sending, etc.) must be considered to achieve better work balance.

Discussion

For parallel Monte Carlo, we need reliable parallel random number generators. The requirements of reproducibility, and hence validation of quality, is important. It isn't clear that selecting different choices of starting seeds for each node will guarantee good randomness properties. The Scalable Parallel Random Number Generators (SPRNG, http://sprng.cs.fsu.edu/) library is one possible candidate. We are working toward incorporating SPRNG into **rpvm** by providing some wrapper functions as well as utilizing existing R functions to generate random numbers from different distributions.

Another challenging problem is to pass higher level R objects through PVM. Because internal data formats may vary across different hosts in the network, simply sending in binary form may not work. Conversion to characters (serialization) appears to be the best solution but there is non-trivial overhead for packing and then sending complicated and/or large objects. This is a similar to the problem of reading in data from files and determining proper data types.

Another future issue is to deploy **rpvm** on Microsoft Windows workstations. Both PVM and R are available under Microsoft Windows, and this is one solution for using additional compute cycles in academic environments.

Bibliography

R. Buyya, editor. *High performance cluster computing: programming and applications, Volume 2*. Prentice Hall, New Jersey, 1999. 6

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing*. MIT Press, Massachusetts, 1994. 4

A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8, 1996. 4

Michael Na Li
University of Washington
lina@u.washington.edu

Anthony J. Rossini
University of Washington
rossini@u.washington.edu