

Package ‘papply’

February 9, 2012

Version 0.1

Date 2005-06-23

Title Parallel apply function using MPI

Author Duane Currie <duane.currie@acadiau.ca>

Maintainer Duane Currie <duane.currie@acadiau.ca>

Depends R (>= 2.0.1)

Suggests Rmpi

Description Similar to apply and lapply, applies a function to all items of a list, and returns a list with the results. Uses Rmpi to distribute the processing evenly across a cluster. If Rmpi is not available, implements this as a non-parallel algorithm. Includes some debugging support

License GPL (>= 2)

URL <http://ace.acadiau.ca/math/ACMMaC/software/papply/>

Repository CRAN

Date/Publication 2005-06-24 05:39:47

R topics documented:

papply 2

Index 6

papply

papply

Description

An apply-like function which uses Rmpi to distribute the processing evenly across a cluster. Will use a non-MPI version if distributed processing is not available.

Usage

```
papply(arg_sets, papply_action, papply_commdata = list(),
       show_errors = TRUE, do_trace = FALSE, also_trace = c())
```

Arguments

arg_sets	a list, where each item will be given as an argument to papply_action
papply_action	A function which takes one argument. It will be called on each element of arg_sets
papply_commdata	A list containing the names and values of variables to be accessible to the papply_action. 'attach' is used locally to import this list.
show_errors	If set to TRUE, overrides Rmpi's default, and messages for errors which occur in R slaves are produced.
do_trace	If set to TRUE, causes the papply_action function to be traced. i.e. Each statement is output before it is executed by the slaves.
also_trace	If supplied an array of function names, as strings, tracing will also occur for the specified functions.

Details

Similar to apply and lapply, applies a function to all items of a list, and returns a list with the corresponding results.

Uses Rmpi to implement a pull idiom in order to distribute the processing evenly across a cluster. If Rmpi is not available, or there are no slaves, implements this as a non-parallel algorithm.

papply will not recursively distribute load. If papply is called within papply_action, it will use a non-parallel version.

The named elements in the list papply_commdata are imported (using 'attach') into a global namespace, and appear as global variables to the code in papply_action.

Value

A list of return values from papply_action. Each value corresponds to the element of arg_sets used as a parameter to papply_action

Note

Does not support distributing recursive calls in parallel. If papply is used inside papply__action, it will call a non-parallel version

Author(s)

Duane Currie duane.currie@acadiu.ca

References

<http://ace.acadiu.ca/math/ACMMaC/software/papply/>

Examples

```
# A couple trivial examples
library(papply)

number_lists <- list(1:10,4:40,2:27)
results <- papply(number_lists,sum)
results

biased_sum <- function(number_list) {
  return(sum(number_list+bias))
}
results <- papply(number_lists,biased_sum,list(bias=2))
results

# A slightly larger example - training of a neural net over a parameter space.
# Produces information on best rss result for each set of parameters.
# Maintains static random seeds in order to provide reproducible results.
# (This isn't ideal. e.g. RSS not the best measure given variation in
# sizes of test sets. But, it should show closely how papply is really used)

# Read in libraries and Boston Housing Data
library(papply)
library(MASS)
data(Boston)

# Generate list of parameter sets
decays <- c(0.2,0.1,0.01)
n_hidden <- c(2,4,6)
parameters <- expand.grid(decays,n_hidden)

# Set random seeds in order to have reproduceable runs.
# 100 is seed for main process which generates folds. Each task will
# have a random seed of the task number
main_seed <- 100
seeds <- c(1:nrow(parameters))

# Create list of argument sets. Each argument is actually a list of
# decay rate, number of hidden nodes, and random seed
```

```

arguments <- list()
for (i in 1:nrow(parameters)) {
  arguments[[i]] <- list(decay=parameters[i,1],
                        hidden=parameters[i,2],
                        seed=seeds[i])
}

# Need to set random seed before generating folds
# Generate random fold labels
set.seed(main_seed)
folds <- sample(rep(1:10,length=nrow(Boston)))

# Make a list of all shared data that should exist in all slaves in the
# cluster
shared_data <- list(folds=folds,Boston=Boston)

# Create function to run on the slave nodes.
# arg is a list with decay, hidden, and seed elements
try_networks <- function(arg) {
  # Make sure nnet library is loaded.
  # NOTE: notice that nnet is not loaded above for the master - it doesn't
  #       need it. It is loaded here because the slaves need it to be
  #       loaded. The is.loaded function is used to test if the nnet
  #       function has been loaded. If not, it loads the nnet library.
  if (!is.loaded("nnet")) {
    library("nnet")
  }

  # Set the random seed to the provided value
  set.seed(arg$seed)

  # Set up a matrix to store the rss values from produced nets
  rss_values <- array(0,dim=c(10,5))

  # For each train/test combination
  for (i in 1:10) {
    # Try 5 times
    for (j in 1:5) {
      # Build a net to predict home value based on other 13 values.
      trained_net <- nnet(Boston[folds!=i,1:13], Boston[folds!=i,14],
                          size=arg$hidden, decay=arg$hidden,
                          linout=TRUE)
      # Try building predictions based on the net generated.
      test <- predict(trained_net, Boston[folds==i,1:13],type="raw")
      # Compute and store the rss values.
      rss <- sqrt(sum((Boston[folds==i,14] - test)^2))
      rss_values[i,j] <- rss
    }
  }

  # Return the rss value of the neural net which had the lowest
  # rss value against predictions on the test set.

```

```
    return(min(rss_values))
  }

# Call the above function for all sets of parameters
results <- papply(arguments, try_networks, shared_data)

# Output a list of parameter vs. minimum rss values
df <- data.frame(decay=parameters[,1],hidden=parameters[,2],
                rss=sapply(results,function(x) {return(x)})
                )
df
```

Index

*Topic **utilities**
papply, [2](#)

papply, [2](#)