Parallel np: Using the npRmpi Package

Jeffrey S. Racine
McMaster University

Abstract

The **npRmpi** package is a parallel implementation of the R (R Development Core Team (2009)) package **np** (Hayfield and Racine (2008)). The underlying C code uses the MPI message passing interface and is MPI2 compliant.

Keywords: nonparametric, semiparametric, kernel smoothing, categorical data..

1. Overview

A common and understandable complaint often levied against nonparametric kernel methods is the large amount of computation time required for data-driven bandwidth selection when one has a large data set. There is a certain irony at play here since nonparametric methods are ideally suited to situations involving large data sets, however computationally speaking, their analysis may lie beyond the reach of many users. Some background may be in order. Cross-validation bandwidth selection methods have run times that are exponential in the number of observations (of computational order n^2 hence a doubling of the sample size will increase run time by a factor of four).

The solution adopted in the **npRmpi** package is to run the code in a parallel computing environment and exploit the presence of multiple processors if available. The underlying C code for **np** is MPI-aware (MPI denotes the 'message passing interface', a popular parallel programming library that is an international standard), and we combine the **R np** and **Rmpi** packages to form the **npRmpi** package (this requires some modification to some of the underlying **Rmpi** code which is why we cannot simply load the **Rmpi** package itself).[1]

All of the functions in **np** can exploit the presence of multiple processors, and for large files run time is in general linear in the number of processors present such that two processors will complete the job in one half the amount of time that one processor could.[2] Given the availability of commodity cluster computers and the presence of multiple cores in desktop and laptop machines, leveraging the **npRmpi** package for large data sets may present a feasible solution to the often lengthy computation times associated with nonparametric kernel methods.

The code has been tested in the Mac OS X and Linux environments which allow the user to

[1] This package incorporates the Rmpi package (Hao Yu <hyu@stats.uwo.ca>) with minor modifications and we are extremely grateful to Hao Yu for his contributions to the R community.

[2] There is minor overhead involved with message passing, and for small samples the overhead can be substantial as the ratio of message passing to computing kernel estimators increases - this will be negligible for sufficiently large samples.

compile R packages on the fly. Users running MS Windows will have to consult their tech support personnel and may also wish to consult the **Rmpi** package and web site for assistance here. I cannot assist with installation issues beyond what is provided in this document and trust the reader will forgive me for this.

2. Differences between np and npRmpi

There are only a few visible differences between running code in serial versus parallel environments. Typically you run your parallel code in batch mode so the first step would be to get your code running in serial mode using the **np** package (obviously on a subset of your data). Once you have properly functioning code, you will next add some 'hooks' necessary for MPI to run, and finally you will run the job using either `mpirun` or, indirectly, via a batch scheduler on your cluster such as `sqsub`.

Installation

Installation will depend on your hardware and software configuration. If you are not familiar with parallel computing you must seek local advice.

That being said, if you have Open MPI and MPI2 properly installed on your system, installation could be as simple as downloading the tarball and, from a command shell, running

```
R CMD INSTALL npRmpi_foo.tar.gz
```

where foo is the version number.

For clusters you may additionally need to provide locations of libraries (kindly see your local sysadmin as there are far too many variations for me to assist). On a local Linux cluster I use the following by way of illustration (we need to set MPI library paths and MPI root directories):

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/sharcnet/openmpi/1.4.1/intel/lib
export MPI_ROOT=/opt/sharcnet/openmpi/1.4.1/intel
R CMD INSTALL npRmpi_foo.tar.gz
```

where again foo is the version number.

Please seek local help for further assistance on installing and running parallel programs.

2.1. Parallel MPI Program Batch Execution

To run the MPI version install the **npRmpi** program, copy the `Rprofile` file in `npRmpi/inst` to the current directory and name it `.Rprofile` (or copy it to your home directory and again name it `.Rprofile`) and then on Open MPI systems run something like

```
mpirun -np 2 R CMD BATCH npudens_npRmpi.R
```

You can compare run times and any other differences by examining the files `npudens_serial.Rout` and `npudens_npRmpi.Rout`. Clearly you could do this with a subset of your data for large problems to judge the extent to which the parallel code reduces run time.

2.2. Essential Program Elements

Here is a simple illustrative example of a serial batch program that you would typically run using the **np** package.

```
## This is the serial version of npudens_npRmpi.R for comparison
## purposes (bandwidth ought to be identical, timing may
## differ). Study the differences between this file and its MPI
## counterpart for insight about your own problems.

library(np)
options(np.messages=FALSE)

## Generate some data

set.seed(42)
x <- rnorm(2500)

## A simple example with likelihood cross-validation

system.time(bw <- npudensbw(~x))

summary(bw)
```

Below is the same code set up to run in parallel using the **npRmpi** package. The salient differences are as follows:

1. You *must* copy the `Rprofile` file from the npRmpi/inst directory of the tarball/zip file into either your root directory or current working directory and rename it `.Rprofile`.

2. You will notice that there are some `mpi.foo` commands where `foo` is, for example, `bcast.cmd`. These the the **Rmpi** commands for telling the slave nodes what to run.

   The first thing we do is initialize the master and slave nodes using the `np.mpi.initialize()` command.

3. Next we broadcast our data to the slave nodes using the `mpi.bcast.Robj2slave()` command which sends an R object to the slaves.

4. After this, we might compute the data-driven bandwidths. Note we have wrapped the **np** command `npudensbw()` in the `mpi.bcast.cmd()` with the option `caller.execute=TRUE` which indicates it is to execute on the master and slave nodes simultaneously.

5. Finally, we clean up gracefully via the `mpi.close.Rslaves()` and `mpi.quit()` commands.

6. There are a number of example files (including that above and below) in the `npRmpi/demo` directory that you may wish to examine. Each of these runs and has been deployed in a range of environments (Mac OS X, Linux).

```
## Make sure you have the .Rprofile file from npRmpi/inst/ in your
## current directory or home directory. It is necessary.

## To run this on systems with OPENMPI installed and working, try
## mpirun -np 2 R CMD BATCH npudens_npRmpi. Check the time in the
## output file foo.Rout (the name of this file with extension .Rout),
## then try with, say, 4 processors and compare run time.

## Initialize master and slaves.

mpi.bcast.cmd(np.mpi.initialize(),
              caller.execute=TRUE)

## Turn off progress i/o as this clutters the output file (if you want
## to see search progress you can comment out this command)

mpi.bcast.cmd(options(np.messages=FALSE),
              caller.execute=TRUE)

## Generate some data and broadcast it to all slaves (it will be known
## to the master node)

set.seed(42)
x <- rnorm(2500)
mpi.bcast.Robj2slave(x)

## A simple example with likelihood cross-validation

system.time(mpi.bcast.cmd(bw <- npudensbw(~x),
                          caller.execute=TRUE))

summary(bw)

## Clean up properly then quit()

mpi.close.Rslaves()

mpi.quit()
```

For more examples including regression, conditional density estimation, and semiparametric models, see the files in the `npRmpi/demo` directory. Kindly study these files and the comments in each in order to extend the parallel examples to your problem.

Note that the output from the serial and parallel runs ought to be identical save for execution time. If they are not there is a problem with the underlying code and I would ask you to kindly report such things to me immediately along with the offending code.

3. Summary

The **npRmpi** package is a parallel implementation of the **np** package that can exploit the presence of multiple processors and the MPI interface for parallel computing to reduce the computational run time associated with kernel methods. Run time is linear in the number of processors available, so two processors will complete a job in roughly one half the time of one processor, ten in one tenth and so forth.[3] Though installation of a working MPI implementation requires some familiarity with computer systems, local expertise exists for many and help is to be found there. That being said, the Mac OS X operating system comes stock with a fully functioning version of Open MPI so there is zero additional effort required for the user in order to get up and running in this environment. Finally, any feedback for improvements for this document, reporting of errors and bugs and so forth is always encouraged and much appreciated.

References

Hayfield T, Racine JS (2008). "Nonparametric Econometrics: The np Package." *Journal of Statistical Software*, **27**(5). URL http://www.jstatsoft.org/v27/i05/.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Affiliation:

Jeffrey S. Racine
Department of Economics
McMaster University
Hamilton, Ontario, Canada, L8S 4L8
E-mail: racinej@mcmaster.ca
URL: http://www.mcmaster.ca/economics/racine/

[3] There is minor overhead involved with message passing, and for small samples the overhead can be substantial as the ratio of message passing to computing kernel estimators increases - this will be negligible for sufficiently large samples.