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 (2010)) package **np** (Hayfield and Racine (2008)). The underlying C code uses the message passing interface ('MPI') and is MPI2 compliant.

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

1. Overview

A common and understandable complaint often voiced about applied nonparametric kernel methods is the 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 yet, computationally speaking, their analysis may lie beyond the reach of many users. Some background may be in order. My co-authors and I favor data-driven methods of bandwidth selection such as cross-validation, among others. These methods possess a number of very desirable properties but have run times that are proportional to the square of the number of observations hence doubling the number of observations will increase run time by a factor of four. For large data sets run time many simply not be feasible in a serial (i.e. single processor) environment.

The solution adopted in the **npRmpi** package is to run the code in a parallel computing environment and exploit the presence of multiple processors when 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 merge the **R np** and **Rmpi** packages to form the **npRmpi** package (this requires minor modification of 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. Run time is inversely proportional to the number of processors hence doubling the number of processors will cut run time in half.[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.

[1]The **npRmpi** 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 when the ratio of message passing to computing the kernel estimator increases - this will be negligible for sufficiently large samples.

The code has been tested in the macOS and Linux environments which allow the user to compile R packages on the fly (presuming of course that a C compiler exists on your system). Users running MS Windows will have to consult local tech support personnel and may also wish to consult the resources available for the **Rmpi** package and associated web site for further assistance. 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 batch mode using the **np** package (obviously on a subset of your data for large data sets). Once you have properly functioning code, you will next add some 'hooks' necessary for MPI to run (see Section 2.3 below for a detailed example), and finally you will run the job using either `mpirun` or, indirectly, via a batch scheduler on your cluster such as `sqsub` (kindly consult your local support personnel for assistance on using batch queueing systems on your cluster).

Note that, since the data has to be broadcast to the slave notes, it is a good idea to put it in a dataframe first and it is always a good idea at this stage to cast your variables according to type.

Installation

Installation will depend on your hardware and software configuration and will vary widely across platforms. If you are not familiar with parallel computing you are strongly advised to seek local advice.

That being said, if you have current versions of R and MPI (e.g., MPICH or Open MPI) properly installed on your system, installation of the **npRmpi** package can be done in the standard manner from within R via

```
> install.packages("npRmpi")
```

or, alternatively, you can download the **npRmpi** tarball from CRAN and, from a command shell, run

```
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 (for this illustration we use the Intel compiler suite and need to set MPI library paths and MPI root directories):

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/sharcnet/mpich/current/intel/lib
export MPI_ROOT=/opt/sharcnet/mpich/current/intel
```

where again foo is the version number.

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

2.1. Parallel Batch Execution

To run a parallel **np** job having successfully installed 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`).[3] Then to run the batch code in the file `npudensml_npRmpi.R` using two processors on an MPI system you will enter something like

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

You can compare run times and any other differences by examining the files `npudensml_serial.Rout` and `npudensml_npRmpi.Rout` (see Section A below for some illustrative examples). 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.

If you have a batch scheduler installed on your cluster you might instead enter something like

```
sqsub -q mpi -n 2 R CMD BATCH npudensml_npRmpi.R
```

Again, kindly consult local tech support personnel for issues concerning the use of batch queueing systems and using compute clusters.

2.2. Parallel Execution in an Interactive R Session

You might instead wish to run your code interactively in an R terminal/console rather than in batch mode. Doing so requires the essential elements in Section 2.3 below with two minor differences. For this we do not require that the initialization file `.Rprofile` reside in the root or current directory (if it does it will be benign). To start a session, (i) you will load the **npRmpi** package in the normal manner via `library("npRmpi")`, then (ii) generate the slave nodes using the command `mpi.spawn.Rslaves(nslaves=foo)`. For example, on a dual core desktop set `foo=1` so that `mpi.spawn.Rslaves(nslaves=1)` will spawn one slave node in addition to the master node on which R is running for a total of two compute nodes. See the file `interactive_Rterm.R` in the demo directory for an illustration, but here is some sample code illustrating the additional elements required for an interactive session.

Note also that in an interactive session the messages displayed during execution might be informative so there is no need for the option `option(np.messages=FALSE)` and you likely would want to comment this statement out in the demo files if you are running them interactively.

```
## This code can be run interactively inside the R terminal/console
## It does not require that you have the .Rprofile file from
## npRmpi/inst/ in your current directory or home directory as we can
## load the npRmpi package in the standard manner.
```

[3] You will need to download the **npRmpi** source code and unpack it in order to get Rprofile from the npRmpi/inst directory.

```
library(npRmpi)

## Now we can spawn our slaves from within the R terminal/console manually.
## On a two core desktop we need an additional slave on top of the master
## node to allow both cores to run simultaneously.

mpi.spawn.Rslaves(nslaves=1)

## Then the rest of your program would follow along the lines of that
## in, say, Section \ref{sec example}...
```

A number of illustrative examples are readily available in the interactive session via `example()` e.g. `example(npreg)` and so forth.

2.3. 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 npudensml_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

n <- 2500

set.seed(42)

x <- rnorm(n)

## A simple example with likelihood cross-validation

t <- system.time(bw <- npudensbw(~x, bwmethod="cv.ml"))

summary(bw)

cat("Elapsed time =", t[3], "\n")
```

Below is the same code modified 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 are 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 by broadcasting the `mpi.quit()` command.

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 (macOS, 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 an MPI implementation (e.g. MPICH)
## installed and working, try
## mpirun -np 2 R CMD BATCH npudensml_npRmpi.R. Check the time in the
## output file npudensml_npRmpi.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)

n <- 2500

mpi.bcast.cmd(set.seed(42), caller.execute=TRUE)

x <- rnorm(n)
mpi.bcast.Robj2slave(x)

## A simple example with likelihood cross-validation
```

```
t <- system.time(mpi.bcast.cmd(bw <- npudensbw(~x, bwmethod="cv.ml"),
                               caller.execute=TRUE))

summary(bw)

cat("Elapsed time =", t[3], "\n")

## Clean up properly then quit()

mpi.bcast.cmd(mpi.quit(), caller.execute=TRUE)
```

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 specific 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 inversely proportional to the number of available processors, so two processors will complete a job in roughly one half the time of one processor, ten in one tenth and so forth.[4] 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 macOS operating system can readily use fully functioning versions of MPI (e.g. MPICH or Open MPI) so there is minimal 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 https://www.jstatsoft.org/v27/i05/.

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL https://www.R-project.org/.

[4] 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 the kernel estimator increases - this will be negligible for sufficiently large samples.

A. Illustrative Timed Runs

The run times reported in Table 1 were generated using R 4.5.2 and MPICH 4.3.2 on an Apple M2 Mac Studio (12 cores) running macOS 15.2 (current date: February 7, 2026). Each example was run in serial mode ($np = 1$) using the **np** package then in parallel mode ($np = 2, 3, 4$) using the **npRmpi** package. Elapsed time for the **np** functions is provided in Table 1 (seconds) as is the ratio of the elapsed time for each parallel run to the serial run.

Table 1: Illustrative timed runs (seconds) with 1 processor (serial, **np** package) and 2, 3, and 4 processors (parallel, **npRmpi** package) on an Apple M2 Mac Studio.

Function	n	Secs(1)	Secs(2)	Secs(3)	Secs(4)	Ratio(2)	Ratio(3)	Ratio(4)
npcdensls	1000	460.6	236.0	166.4	126.0	0.51	0.36	0.27
npcdensml	2500	35.5	18.0	12.2	9.4	0.51	0.34	0.27
npcdistls	2000	84.1	43.2	36.7	31.2	0.51	0.44	0.37
npcmstest	616	10.0	5.8	4.4	3.8	0.58	0.44	0.38
npconmode	189	8.9	5.3	4.1	3.5	0.60	0.47	0.40
npcopula	5000	5.1	3.0	3.1	2.5	0.60	0.61	0.50
npdeneqtest	2500	33.8	17.1	11.9	9.3	0.50	0.35	0.28
npdeptest	2500	41.9	21.1	14.7	11.3	0.50	0.35	0.27
npglpreg	1500	422.6	235.7	167.7	136.3	0.56	0.40	0.32
npindexich	5000	18.6	10.2	6.8	5.2	0.55	0.37	0.28
npindexks	5000	24.6	13.4	9.0	6.9	0.54	0.37	0.28
npplreg	1000	10.1	6.0	4.3	3.4	0.59	0.42	0.34
npqreg	1008	27.1	15.1	12.2	10.2	0.56	0.45	0.38
npregiv	2500	139.1	112.2	97.5	91.5	0.81	0.70	0.66
npreglcaic	5000	160.8	82.9	55.0	40.4	0.52	0.34	0.25
npreglcls	5000	157.1	81.0	54.5	39.7	0.52	0.35	0.25
npregllaic	5000	97.0	66.1	50.5	39.3	0.68	0.52	0.40
npregllls	5000	96.7	64.1	48.9	38.4	0.66	0.51	0.40
npscoef	10000	42.7	24.6	17.9	14.7	0.58	0.42	0.34
npsdeptest	1500	70.8	37.3	26.4	20.8	0.53	0.37	0.29
npsigtest	1000	56.9	42.9	47.1	37.9	0.75	0.83	0.67
npsymtest	2500	38.9	20.4	14.5	11.1	0.52	0.37	0.28
npudensls	10000	85.1	42.0	28.2	21.2	0.49	0.33	0.25
npudensml	10000	42.5	20.9	14.0	10.7	0.49	0.33	0.25
npudistcdf	10000	119.5	62.6	89.3	78.7	0.52	0.75	0.66
npunitest	5000	155.5	76.4	52.2	39.5	0.49	0.34	0.25

Note that many of these illustrative examples use smallish sample sizes hence the run time with 2 processors will not be 1/2 that with 1 processor due to overhead. But for larger samples (i.e. the ones you actually need parallel computing for, not these toy illustrations) you ought to see an improvement in run time that is inversely related to the number of processors.

Affiliation:

Jeffrey S. Racine
Department of Economics
McMaster University
Hamilton, Ontario, Canada, L8S 4L8
E-mail: racinej@mcmaster.ca
URL: https://experts.mcmaster.ca/people/racinej