Scripting with R in high-performance computing: An Example using littler

UseR! 2008 conference

Dirk Eddelbuettel

TU Dortmund
August 13, 2008

Abstract

Abstract

High-Performance Computing with R often involves distributed computing. Here, the MPI toolkit is a popular choice, as it is well supported in R by the `Rmpi` and `snow` packages. In addition, resource and and queue managers like slurm help in allocating and managing computational jobs across compute nodes and clusters.

In order to actually to execute tasks, we can take advantage of a scripting frontend to R such as `r` (from the `littler` package) or `Rscript`. By discussing a stylized yet complete example, we will provide details about how to organise a task for R by showing how to take advantage of automated execution across a number of compute nodes while being able to monitor and control its resource allocation.

High-performance computing with R

- Several possible definitions of High-Performance Computing (HPC) with R
- Some of those were discussed in the introductory 'HPC with R' tutorial on Monday
- Here we are focussing on *parallel computing* using the MPI toolkit
- as well as the *Rmpi* and *snow* packages for R
- and the *slurm* resource allocation / batch / queueing sytem that works well with MPI
- and how using *R scripting* fits in rather nicely with this framework.

Scripting with R ?

Being able to launch numerous R jobs in a parallel environments is helped by the ability to 'script' R.

Several simple methods existed to start R:

- `R CMD BATCH file.R`
- `echo "commands" | R -no-save`
- `R -no-save < file.R > file.Rout`

These are suitable for one-off scripts, but may be too fragile for distributed computing.

Scripting with r !

The `r` command of the `littler` package (as well as R's `Rscript`) provide more robust alternatives.

`r` can also be used four different ways:

- `r file.R`
- `echo "commands" | r`
- `r -lRmpi -e 'cat("Hello", mpi.get.processor.name())'`
- and *shebang*-style in script files: `#!/usr/bin/r`

It is the last point that is of particular interest in this HPC context.

Also of note is the availability of the `getopt` package on CRAN.

Rmpi

`Rmpi` is a CRAN package that provides and interface between R and the Message Passing Interface (MPI), a standard for parallel computing. (c.f. Wikipedia for more and links to the Open MPI and MPICH2 projects for implementations).

The preferred implementation for MPI is now Open MPI. However, the older LAM implementation can be used on those platforms where Open MPI is unavailable. There is also an alternate implementation called MPICH2.

`Rmpi` allows us to use MPI directly from R .

MPI Example

Let us look at the `MPI` variant of the standard 'Hello, World!' program:

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char** argv)
5  {
6      int rank, size, nameLen;
7      char processorName[MPI_MAX_PROCESSOR_NAME];
8
9      MPI_Init(&argc, &argv);
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13     MPI_Get_processor_name(processorName, &nameLen);
14
15     printf("Hello, rank %d, size %d on processor %s\n",
16            rank, size, processorName);
17
18     MPI_Finalize();
19     return 0;
20 }
```

Rmpi

`Rmpi` wraps many of the MPI API calls for use by R, so the preceding example can be rewritten in R as

```
1 #!/usr/bin/env r
2
3 library(Rmpi) # calls MPI_Init
4
5 rk <- mpi.comm.rank(0)
6 sz <- mpi.comm.size(0)
7 name <- mpi.get.processor.name()
8 cat("Hello, rank", rk, "size", sz, "on", name, "\n")
```

Or for that matter:

```
$ r -lRmpi -e'cat("Hello", \
      mpi.comm.rank(0), "of", \
      mpi.comm.size(0), "on", \
      mpi.get.processor.name(), "\n")'
```

Running code under (Open) MPI typically involves calling `orterun`, the replacement for the `mpirun` wrapper.

slurm resource management and queue system

Once the number of compute nodes increases, it becomes of interest to be able to allocate and manage resources, and to queue and batch jobs. A suitable tool is `slurm`, an open-source resource manager for Linux clusters.

Paraphrasing from the slurm website:

- it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users;
- it provides a framework for starting, executing, and monitoring (typically parallel) work on a set of allocated nodes.
- it arbitrates contention for resources by managing a queue of pending work.

Slurm is being developed by a consortium including LLNL, HP, Bull, and Linux Networks.

slurm example

Slurm wraps around Open MPI. That permits use of `Rmpi` and other recent MPI-using applications built against Open MPI.

```
$ srun -n 4 -N 2 -O r -lRmpi -e'cat("Hello", \
mpi.comm.rank(0), "of", \
        mpi.comm.size(0), "on", \
        mpi.get.processor.name(), "\n")'

Hello 0 of 1 on ron
Hello 0 of 1 on ron
Hello 0 of 1 on joe
Hello 0 of 1 on joe
```

In this example using `srun`, we use the `-O` overcommit option to launch four jobs on the two nodes available.

slurm and snow

We would like to use `snow` with `slurm` as well.

However, there is are problems:

- `snow` has a master/worker paradigm yet `slurm` launches its nodes symmetrically,
- `slurm`'s `srun` has limits in spawning jobs
- with `srun`, we cannot communicate the number of nodes 'dynamically' into the script: `snow`'s cluster creation needs a hardwired number of nodes

slurm and snow solution

`snow` solves the master / worker problem by auto-discovery upon startup. The package contains two internal files `RMPISNOW` and `RMPISNOWprofile` that use a combination of shell and R code to determine the node idendity allowing it to switch to master or worker functionality.

We can reduce the same problem to this for our R script:

```
ndsvpid <- Sys.getenv("OMPI_MCA_ns_nds_vpid")
if (ndsvpid == "0") {                   # are we the master ?
    makeMPIcluster()
} else {                                # or are we a slave ?
    sink(file="/dev/null")
    slaveLoop(makeMPImaster())
    q()
}
```

salloc for snow

The other important part is to switch to `salloc` (as well as `orterun`) instead of `srun`.

We can either supply the hosts used using the `-w` switch, or rely on the `slurm.conf` file.

But importantly, we can govern from the call how many instances we want running (and have neither the `srun` limitation nor the hard-coded `snow` cluster-creation size):

```
$ salloc -w ron,mccoy orterun -n 7 rMPIsnow.r
```

We ask for a `slurm` allocation on the given hosts, and instruct the Open MPI to run seven instances.

A complete example

```
cl <- NULL
ndsvpid <- Sys.getenv("OMPI_MCA_ns_nds_vpid")
if (ndsvpid == "0") {                          # are we master ?
    cl <- makeMPIcluster()
} else {                                       # or are we a slave ?
    sink(file="/dev/null")
    slaveLoop(makeMPImaster())
    q()
}

clusterEvalQ(cl, library(RDieHarder))
res <- parLapply(cl, c("mt19937", "mt19937_1999",
                       "mt19937_1998", "R_mersenne_twister"),
                 function(x) {
                     dieharder(rng=x, test="operm5",
                               psamples=100, seed=12345,
                               rngdraws=100000)
                     }
                 )
stopCluster(cl)
```

A complete example cont.

This uses `RDieHarder` to test four Mersenne-Twister implementations at once.

A simple analysis shows the four charts and prints the four *p*-values:

```
pdf("/tmp/snowRDH.pdf")
lapply(res, function(x) plot(x))
dev.off()

print( do.call(rbind, lapply(res, function(x) { x[[1]] } )))
```

Summary

We have seen

- how `littler` can help us script R tasks
- how `Rmpi`, `snow` and `slurm` can interact nicely
- a complete example using `RDieHarder` to illustrate these concepts

Last but not least: all software is in Debian as well as on a UseR! 2008 live cdrom at `http://quantian.alioth.debian.org`.