An introduction to distributed memory parallelism in R and C

Chris Paciorek

January 4, 2013

Note: my examples here will be silly toy examples for the purpose of keeping things simple and focused on the parallelization approaches. The syntax here is designed running jobs on SCF Linux machines but analogous code should work on other clusters and in the cloud, such as Amazon's EC2.

1 Overview of parallel processing

There are two basic flavors of parallel processing (leaving aside GPUs): distributed memory and shared memory. With shared memory, multiple processors (which I'll call cores) share the same memory. With distributed memory, you have multiple nodes, each with their own memory. Each node generally has multiple cores. You can think of each node as a separate computer connected by a fast network to the other nodes.

1.1 Distributed memory

Parallel programming for distributed memory parallelism requires passing messages between the different nodes. The standard protocol for doing this is MPI, of which there are various versions, including *openMPI* and *mpich*. The R package *Rmpi* implements MPI in R.

1.2 Shared memory

For shared memory parallelism, each core is accessing the same memory so there is no need to pass messages. But one needs to be careful that activity on different cores doesn't mistakenly overwrite places in memory that are used by other cores. For more details on shared memory computing, please see my tutorial, Introduction to Parallel Programming for Statisticians.

2 Parallel processing in the SCF

At the moment, we are configured in the following way for parallel processing.

1. **Shared memory**: One can do shared memory parallelism (1) directly on the compute servers or (2) on the cluster by submitting jobs through *qsub*. On the compute servers we ask that you keep these guidelines in mind to fairly share the resources with others. Cluster jobs can use up to 32 cores, all on a single node; we have not enabled the use of multiple nodes or more than 32 cores for a single job, but if you have need of this, please contact consult@stat.berkeley.edu.

2. **Distributed memory**: One can do distributed memory parallelism on the compute servers as described later in this document. On the compute servers we ask that you keep these guidelines in mind to fairly share the resources with others.

3. **Hadoop**: We have a test installation of hadoop running on 8 old compute servers. More information can be found here.

3 MPI basics

There are multiple MPI implementations, of which *openMPI* and *mpich* are very common.

In MPI programming, the same code runs on all the machines, but each machine is assigned a unique identifying number, so the code can include logic so that different code paths are followed on different machines. Since MPI operates in a distributed fashion, any transfer of information between machines must be done explicitly via send and receive calls (*MPI_Send*, *MPI_Recv*, *MPI_Isend*, and *MPI_Irecv*).

The latter two of these functions (*MPI_Isend* and *MPI_Irecv*) are so-called non-blocking calls. One important concept to understand is the difference between blocking and non-blocking calls. Blocking calls wait until the call finishes, while non-blocking calls return and allow the code to continue. Non-blocking calls can be more efficient, but can lead to problems with synchronization between processes.

4 Using message passing on a single node

MPI (specifically *openMPI*) is installed on all the Linux compute servers, including the cluster nodes. R users can use *Rmpi* to interface with MPI or can use the functionality in the *parallel* package to make a *socket cluster*, which doesn't use MPI, but does its own version of message passing. There's not much reason to use MPI on a single node as there's a cost to the message passing relative to shared memory, but it is useful to be able to test the code on a single machine without having to worry about networking issues across nodes.

MPI example There are C (*mpicc*) and C++ (*mpic++*, *mpicxx*, *mpiCC* are synonyms on the SCF system) compilers for MPI programs.

Here's a basic hello world example (I'll use the MPI C++ compiler even though the code is all plain C code).

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

int main(int argc, char** argv) {
int myrank, nprocs, namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Get_processor_name(processor_name, &namelen);
printf("Hello from processor %d of %d on %s\n",
myrank, nprocs, processor_name);
    MPI_Finalize();
return 0;
}
```

To compile and run the code, do:

```
mpicxx mpiHello.c -o mpiHello
mpirun -np 4 mpiHello
```

This will run multiple (four in this case) processes on the machine on which it is run. Here's the output:

```
Hello from processor 2 of 4 on arwen
Hello from processor 3 of 4 on arwen
Hello from processor 0 of 4 on arwen
Hello from processor 1 of 4 on arwen
```

Note that *mpirun*, *mpiexec*, and *orterun* are synonyms under *openMPI*.

To actually write real MPI code, you'll need to go learn some of the MPI syntax. The accompanying zip file has example C and C++ programs (for approximating an integral via quadrature) that show some of the basic MPI functions. Compilation and running are as above:

```
mpicxx quad_mpi.cpp -o quad_mpi
mpirun -np 4 quad_mpi
```

And here's the output:

```
03 January 2013 04:02:47 PM

QUAD_MPI
C++/MPI version
Estimate an integral of f(x) from A to B.
f(x) = 50 / (pi * ( 2500 * x * x + 1 ) )

A = 0
B = 10
N = 999999999
EXACT =       0.4993633810764567
Use MPI to divide the computation among 4 total processes,
of which one is the master and does not do core computations.
Process 2 contributed MY_TOTAL = 0.00095491
Process 1 contributed MY_TOTAL = 0.49809
Process 3 contributed MY_TOTAL = 0.000318308

Estimate =       0.4993634591634721
Error = 7.808701535383378e-08
Time = 9.904016971588135

QUAD_MPI:
Normal end of execution.

03 January 2013 04:02:56 PM
```

Note that *mpirun* just dumbly invokes as many processes as you request without regard to the hardware (i.e., the number of cores) on your machine.

Here's how we would run the program on the SCF cluster via *qsub*. One would often put the command passed to *qsub* inside a shell script but here I just do it at the command line.

```
np=4; qsub -pe smp $np -b y 'mpirun -np $np quad_mpi >& results'
```

Rmpi example Simply start R as you normally do.

```
R
```

or

```
R CMD BATCH --no-save example.R example.out
```

Here's R code for using *Rmpi* as the back-end to *foreach*, which is a common way of parallelizing for loops.

```
# basic example with foreach

# start R as usual: 'R' or via a batch job
library(Rmpi)
library(doMPI)
nslaves <- 4
cl <- startMPIcluster(nslaves)

##  4 slaves are spawned successfully. 0 failed.

registerDoMPI(cl)
clusterSize(cl)  # just to check

## [1] 4


result <- foreach(i = 1:20) %dopar% {
    out <- mean(rnorm(1e+07))
}

closeCluster(cl)

# you can also start as
#'mpirun -np 1 R --no-save'
#'mpirun -np 1 R CMD BATCH --no-save example.R example.out'

# if you do this, you should quit via: mpi.quit()
```

One can also call *mpirun* to start R. Here one requests just a single process, as R will manage the task of spawning slaves.

```
mpirun -np 1 R --no-save
```

or

```
mpirun -np 1 R CMD BATCH --no-save example.R example.out
```

Note that running R this way is clunky as the usual tab completion and command history are not available and errors cause R to quit, because passing things through *mpirun* causes R to think it is not running interactively. Given this, at the moment it's not clear to me why you would want to invoke R using *mpirun*, but I include it for completeness.

Here's some example code that uses actual *Rmpi* syntax, which is very similar to the MPI C syntax we've already seen. This code runs in a master-slave paradigm where the master starts the slaves and invokes commands on them. It may be possible to run *Rmpi* in a context where each process runs the same code based on invoking with Rmpi, but I haven't investigated this further.

```
# example syntax of standard MPI functions

library(Rmpi)
mpi.spawn.Rslaves(nslaves = 4)

##  4 slaves are spawned successfully. 0 failed.
## master (rank 0, comm 1) of size 5 is running on: smeagol
## slave1 (rank 1, comm 1) of size 5 is running on: smeagol
## slave2 (rank 2, comm 1) of size 5 is running on: smeagol
## slave3 (rank 3, comm 1) of size 5 is running on: smeagol
## slave4 (rank 4, comm 1) of size 5 is running on: smeagol


n <- 5
mpi.bcast.Robj2slave(n)
mpi.bcast.cmd(id <- mpi.comm.rank())
mpi.bcast.cmd(x <- rnorm(id))

mpi.remote.exec(ls(.GlobalEnv), ret = TRUE)

## $slave1
## [1] "id" "n"  "x"
##
```

```
## $slave2
## [1] "id" "n"  "x"
##
## $slave3
## [1] "id" "n"  "x"
##
## $slave4
## [1] "id" "n"  "x"
```

```
mpi.bcast.cmd(y <- 2 * x)
mpi.remote.exec(print(y))
```

```
## $slave1
## [1] 1.923
##
## $slave2
## [1]  1.186 -1.255
##
## $slave3
## [1]  2.5489  2.7242 -0.9396
##
## $slave4
## [1]  0.4350  0.9240 -0.2448  0.4019
```

```
objs <- c("y", "z")
# next command sends value of objs on _master_ as
# argument to rm
mpi.remote.exec(rm, objs)
```

```
## $slave2
## [1] 0
```

```
mpi.remote.exec(print(z))
```

```
## $slave1
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>
##
## $slave2
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>
##
## $slave3
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>
##
## $slave4
## [1] "Error in print(z) : object 'z' not found\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in print(z): object 'z' not found>


# collect results back via send/recv
mpi.remote.exec(mpi.send.Robj(x, dest = 0, tag = 1))

## $slave2
## [1] 0
##
## $slave4
## [1] 0

results <- list()
```

```
for (i in 1:(mpi.comm.size() - 1)) {
    results[[i]] <- mpi.recv.Robj(source = i, tag = 1)
}

print(results)

## [[1]]
## [1] 0.9617
##
## [[2]]
## [1]  0.5929 -0.6274
##
## [[3]]
## [1]  1.2745  1.3621 -0.4698
##
## [[4]]
## [1]  0.2175  0.4620 -0.1224  0.2010
```

A caution concerning Rmpi/doMPI: when you invoke *startMPIcluster()*, all the slave R processes become 100% active and stay active until the cluster is closed. In addition, when *foreach* is actually running, the master process also becomes 100% active. So using this functionality involves some inefficiency in CPU usage. This inefficiency is not seen with a sockets cluster (see next) nor when using other Rmpi functionality - i.e., starting slaves with *mpi.spawn.Rslaves()* and then issuing commands to the slaves.

Sockets in R example One can also set up a cluster via sockets. In this case we'll do it on a remote machine but put all the workers on that same machine.

```
# example with PSOCK cluster

library(parallel)

machine <- "arwen"
cl <- makeCluster(rep(machine, 4))
# cl = makeCluster(4) # this would do it by default on
# your local machine
n <- 1e+07
```

```
clusterExport(cl, c("n"))
fun <- function(i) out <- mean(rnorm(n))

result <- parSapply(cl, 1:20, fun)

stopCluster(cl)  # not strictly necessary
```

One reason to use Rmpi on a single node is that for machines using openBLAS (all of the compute servers except for the cluster nodes), there is a conflict between the multicore/parallel's forking functionality and openBLAS that causes foreach to hang when used with the doParallel or doMC parallel back ends.

Threaded linear algebra Note that if you use linear algebra in your R calculations (or if you run an *openMP* program via *mpirun*), you will end up using more than the number of cores specified by the number of slaves requested. To control this you can set OMP_NUM_THREADS before starting your program. However if you are running on a remote machine, such as the sockets example, this variable would not get propagated to the other machines, unless you had set it (say, temporarily) in your *.bashrc* file. At the moment I don't know of any other way to control this.

5 Using message passing in a distributed memory environment

MPI example To run on multiple machines, we need to let mpirun know the names of those machines. We can do this in two different ways.

First, we can pass the machine names directly, replicating the name if we want multiple processes on a single machine.

```
mpirun --host arwen,arwen,treebeard -np 3 mpiHello
```

Here's the output:

```
Hello from processor 0 of 3 on arwen
Hello from processor 2 of 3 on treebeard
Hello from processor 1 of 3 on arwen
```

```
mpirun --host arwen,arwen,treebeard -np 3 quad_mpi
```

Alternatively, we can create a file with the relevant information. Here we'll specify two processes on arwen for every one process on treebeard.

```
echo 'arwen slots=2' >> .hosts
echo 'treebeard slots=1' >> .hosts
```

```
mpirun -hostfile .hosts -np 3 mpiHello
mpirun -hostfile .hosts -np 12 mpiHello # this seems to just recycle
what is in .hosts
```

Here's the output from the last command that requested 12 processes:

```
Hello from processor 1 of 12 on arwen
Hello from processor 9 of 12 on arwen
Hello from processor 10 of 12 on arwen
Hello from processor 3 of 12 on arwen
Hello from processor 0 of 12 on arwen
ello from processor 6 of 12 on arwen
Hello from processor 7 of 12 on arwen
Hello from processor 4 of 12 on arwen
Hello from processor 5 of 12 on treebeard
Hello from processor 8 of 12 on treebeard
Hello from processor 11 of 12 on treebeard
Hello from processor 2 of 12 on treebeard
```

To limit the number of threads for each process, we can tell *mpirun* to export the value of OMP_NUM_THREADS to the processes.

```
export OMP_NUM_THREADS=2
mpirun -hostfile .hosts -np 3 -x OMP_NUM_THREADS quad_mpi
```

Rmpi example Here for interactive use we run *mpirun* with -np 1 but provide the hosts information, so *Rmpi* can make the requisite connections for the worker processes. In my testing it appears that *Rmpi* wants the first host in the hosts file to be the machine on which you run *mpirun*.

```
mpirun -hostfile .hosts -np 1 R --no-save
```

Our host file specifies three slots so I'll ask for two slaves in the R code, since there is also the master process and it will (inefficiently) use up the resources on a core. But note that whatever is in .hosts will get recycled if you ask for more processes than the number of slots.

```
# start R as either: mpirun -hostfile .hosts -np 1 R
# --no-save mpirun -hostfile .hosts -np 1 R CMD BATCH
# --no-save
library(Rmpi)
library(doMPI)
nslaves <- 2
```

```
# 2 slaves since my host file specifies 3 slots, and
# one will be used for master
cl <- startMPIcluster(nslaves)
registerDoMPI(cl)
clusterSize(cl)  # just to check
results <- foreach(i = 1:200) %dopar% {
    out <- mean(rnorm(1e+07))
}
if (FALSE) {
    foreach(i = 1:20) %dopar% {
        out <- chol(crossprod(matrix(rnorm(3000^2), 3000)))[1,
            2]
    }
}
closeCluster(cl)
mpi.quit()
```

For a batch job, you can do the same as above, but of course with R CMD BATCH. Alternatively, you can do the following with $np set to a number greater than one:

```
mpirun -hostfile .hosts -np $np R CMD BATCH --no-save tmp.R tmp.out
```

Then in your R code if you call startMPIcluster() with no arguments, it will start up $np-1 slave processes by default, so your R code will be more portable.

If you specified -np with more than one process then as with the C-based MPI job above, you can control the threading via OMP_NUM_THREADS and the -x flag to *mpirun*. Note that this only works when the R processes are directly started by *mpirun*, which they are not if you set -np 1. The *maxcores* argument to *startMPIcluster()* does not seem to function (perhaps it does on other systems).

Also note that if you do this in interactive mode, some of the usual functionality of command line R (tab completion, scrolling for history) is not enabled and errors will cause R to quit. This occurs because passing things through *mpirun* causes R to think it is not running interactively.

Note on supercomputers Note: in some cases a cluster/supercomputer will be set up so that *Rmpi* is loaded and the worker processes are already started when you start R. In this case you wouldn't need to load *Rmpi* or use *mpi.spawn.Rslaves()* (though you would still need to run *startMPIcluster()* if using *foreach* with *doMPI*). You can always check *mpi.comm.size()* to see if the workers are already set up.

Sockets in R example This is not much different from the single node case. You just need to specify a character vector with the machine names as the input to *makeCluster()*.

```
# multinode example with PSOCK cluster

library(parallel)

machineVec <- c(rep("arwen", 4), rep("treebeard", 2), rep("beren",
    2))
cl <- makeCluster(machineVec)

n <- 1e+07
clusterExport(cl, c("n"))
fun <- function(i) out <- mean(rnorm(n))

result <- parSapply(cl, 1:20, fun)

stopCluster(cl)  # not strictly necessary
```

As before, be aware that if you use linear algebra, your processes will use more cores than the number of slaves specified.

6 Debugging and other issues

Debugging MPI/Rmpi code can be tricky because communication can hang, error messages from the workers may not be seen or readily accessible and it can be difficult to assess the state of the worker processes.

As is generally true with parallel processing one needs to be careful about random number generation. See my notes in Introduction to Parallel Programming for Statisticians.