Parallel R

M. T. Morgan (mtmorgan@fhcrc.org)

Fred Hutchinson Cancer Research Center
Seattle, WA

http://bioconductor.org

4 August, 2006

Introduction

Why parallel?

- Long computations, or big data.
- Goal is to divide computation burden among processors.

Solutions with R

- Usually, no 'magic bullet'
  - R is not *thread safe*, all data is *in memory*.
  - Algorithms are written for serial processing.
  - Not quite true, e.g., BLAS/LAPACK[a] libraries.
- Instead: *ad hoc* solutions with packages such as Rmpi.

[a] http://cran.fhcrc.org/doc/manuals/R-admin.html

Problems we will touch on

- Random number generation. Useful to introduce methods and suggest challenges.
- Interactively exploring `ExpressionSet` data.
- Cross-validation. *Embarassingly parallel*: each cross-validation independent of other iterations. Readily parallelized.
- (Advanced) Bootstrapping. Also embarassingly parallel, but more work required to parallelize.
- (Advanced) Interfacing C code.

Solutions and limitations

- Write an R script for sequential processing; identify bottlenecks in code execution.
- Modify to allow parallelization.
  - Load packages such as Rmpi.
  - Redefine or modify functions to distribute calculations.
- Often: develop algorithm interactively, evaluate in `BATCH` mode.

Limitations.

- Quite a bit of programming required.
- Maximum speedup limited by fraction of parallelizable code.
- Communication costs suggest *coarse-grained* parallelization.
- Diminishing proportional benefits of additional processors.

A first session

```
> library(Rmpi)
> mpi.spawn.Rslaves(nslaves = 2)

        2 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 3 is running on: gladstone
slave1 (rank 1, comm 1) of size 3 is running on: gladstone
slave2 (rank 2, comm 1) of size 3 is running on: gladstone
```

- One *node* (*master* or *manager*) coordinates tasks, other nodes (*slaves* or *workers*) perform computations.
- `nslaves` can be more or less than the number of processing units (e.g., CPUs) available.
- Each spawned R is a separate process, sharing only a mechanism of communication with the other R processes.

Sending data and commands

```
> x <- 1:5
> mpi.bcast.Robj2slave(x)
```

- Efficiently send (*broadcast*) any R object (including functions).
  - Internally: using `serialize`.

```
> mpi.remote.exec(search())[1]
```

```
$slave1
 [1] ".GlobalEnv"        "package:Rmpi"
 [3] "package:methods"   "package:stats"
 [5] "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"
 [9] "Autoloads"         "package:base"
```

- Evaluate and receive results of any R function.

Calculations on `ExpressionSet`

```
> library(golubEsets)

Loading required package: Biobase
Loading required package: tools

> data(golubMerge)
> exprSet <- exprs(golubMerge)
> res <- apply(exprSet, 1, mad)
> res <- mpi.parApply(exprSet, 1, mad)
```

- `mpi.parApply` like `apply`, but first argument divided between workers.
- Expensive communication, because portions of `exprSet` sent 'over the wire' to each worker.

Another way...

```
> ff <- function(i) mad(exprSet[i, ])
> res <- sapply(1:nrow(exprSet), ff)
```

Parallel code:

```
> mpi.bcast.cmd(library(golubEsets))
> mpi.bcast.cmd(data(golubMerge))
> mpi.bcast.cmd(exprSet <- exprs(golubMerge))
> res <- mpi.parSapply(1:nrow(exprSet), ff)
```

- `mpi.bcast.cmd` like `mpi.remote.exec`, but no return value.
- Data loaded from local disk.
- `mpi.parSapply` like `sapply`; `FUN` sent to workers.
- Only a vector of length 7129 sent and received.

Random numbers

```
> mpi.remote.exec(runif(4))
```

```
          X1        X2
1 0.8830365 0.8830365
2 0.6278537 0.6278537
3 0.6288069 0.6288069
4 0.1779682 0.1779682
```

- Not very random!
- Each node has the same random number seed, so creates the same random number sequence.
- Parallel computaton, but not very useful: apply the same *program* to the same *data*.

```
> mpi.setup.rngstream()

Loading required package: rlecuyer

> mpi.remote.exec(runif(4))

         X1        X2
1 0.1507135 0.5593815
2 0.5115667 0.4183043
3 0.9870720 0.3926978
4 0.1850241 0.6008186
```

Repeatable research can be *very* problematic

- Identical results require repeatable random number sequence *and* identical order of evaluation.
- Order of evaluation depends on, e.g., cluster size, but also vaguaries of processs timing.

Lab, part 1

Next: toward useful work

Cross-validation and machine learning

Can gene expression patterns help identify phenotype?

- Divide known phenotypes into a 'training' and 'test' set.
- Train a machine learning algorithm with the training set.
- Test the trained alogrithm (comparing predicted and known phenotypes) with the 'test' set.
- Cross-validiation: repeat with other training and test sets.
- Select 'best' machine learning algorithm.

Cross-validiation

- Statistically assess machine learning algorithm.
- Each cross-validiation (almost) independent.

An example: `golubMerge` data set

```
> library(MLInterfaces)
> library(golubEsets)
> data(golubMerge)
```

- Data set of 7129 gene expression values measured on 72 samples.
- 11 phenotypic measures on each sample, including leukemia status (`ALL` or `AML`).

We will look at a subset of the data:

```
> smallG <- golubMerge[200:250, ]
```

Cross-validation

```
> lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
+     group = as.integer(0))
> table(lk1, smallG$ALL.AML)
```

```
lk1   ALL AML
  ALL  37  10
  AML  10  15
```

- Classify patient leukemia status using `knnB` algorithm (k nearest neighbors).
- `xvalMethod`: `leave-one-out` – training set is all but one sample, testing set the remaining sample.
- Cross-validate with all possible training and test sets.
- Interpretation: 72 cross-validations, 52 correct classifications.

Cross-validation in parallel

```
> mpi.bcast.cmd(library(MLInterfaces))
> lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
+     group = as.integer(0), cluster = cluster)
> table(lk1, smallG$ALL.AML)

lk1   ALL AML
  ALL  37  10
  AML  10  15
```

- Same results as before (good!)
- MLInterfaces package developers modified `xval` for easy parallelization.
- Implementation presented here has high communication costs, so does not scale too well.

Under the hood...

```
> setClass("RmpiXval", representation("list"))
> setMethod("xvalLoop", signature(cluster = "RmpiXval"),
+     function(cluster, ...) mpi.parLapply)
> cluster = new("RmpiXval")
```

So far...

- Start a single process, spawn several workers, distribute data, do analysis, return result.

Room for improvement.

- Interactive.
- Confusing mix of standard and parallel code.
- High communication costs to distribute data.
- Cluster configuration inside R.
- 'Manager' never does any real work.

Batch programing

Write a script file `xval-batch.R`...

```
# file xval-batch.R
# Load Rmpi, setup RmpiXval, xvalLoop (details above)

# broadcast and analyze
mpi.bcast.cmd(library(MLInterfaces))
library(MLInterfaces)
library(golubEsets)
data(golubMerge)
smallG <- golubMerge[200:250, ]
lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
            group = as.integer(0), cluster=new("RmpiXval"))
table(lk1, smallG$ALL.AML)
```

. . . and execute (from the command line) in 'batch' mode.

```
% R CMD BATCH xval-batch.R
```

Output presented in `xval-batch.Rout`.

- Original issues:
  - Interactive. (SOLVED)
  - Confusing mix of standard and parallel code. (SOLVED?)
  - High communication costs to distribute data.
  - Cluster configuration inside R.
  - 'Manager' never does any real work.

A different parallel style

- Often, each node has its own hard drive, and cluster hardware efficiently moves large data to each drive. So...
- Each node determines data to analyze, performs the analysis, and coordinates the (usually much smaller) results with other nodes.

```
# file xval-batch-2.R
# Load Rmpi, setup RmpiXval2, xvalLoop (details elsewhere)

# analyze data
library(MLInterfaces)
library(golubEsets)
data(golubMerge)
smallG <- golubMerge[200:250, ]
lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod = "LOO",
```

```
            group = as.integer(0), cluster=new("RmpiXval2"))
# output results, but only once!
if (mpi.comm.rank() == 0)
   table(lk1, smallG$ALL.AML)
```

Control cluster (e.g., using 3 nodes) from the command line:

```
% mpiexec -n 3 R CMD BATCH xval-batch-2.R
```

Original issues:

- Interactive. (SOLVED)
- Confusing mix of standard and parallel code. (SOLVED)
- High communication costs to distribute data. (SOLVED)
- Cluster configuration inside R. (SOLVED)
- 'Manager' never does any real work. (SOLVED)

Under the hood...

```
> setClass("RmpiXval2", representation = list(size = "numeric"))
> setMethod("xvalLoop", signature(cluster = "RmpiXval2"),
+     function(cluster, ...) lapplys)
> cluster <- new("RmpiXval2")
```

- All nodes start `xval`, locally prepare data for analysis.
- `xvalLoop` method partitions work for each node, allows computation to occur in parallel, collates results.
- All nodes massage and return result.

Really under the hood: `allgather.Robj`

Efficiently collate `obj` from all nodes.

```
> allgather.Robj <- function(obj = NULL, comm = 1) {
+     obj <- as.integer(charToRaw(serialize(obj,
+         NULL)))
+     sz <- mpi.allgather(length(obj), 1, integer(mpi.comm.size(c
+         comm)
+     objs <- mpi.allgatherv(obj, 1, integer(sum(sz)),
+         sz, comm)
+     as.list(unlist(lapply(split(as.raw(objs),
+         rep(1:length(sz) - 1, sz)), unserialize),
+         recursive = FALSE, use.names = FALSE))
+ }
```

Really under the hood: `lapplys`

`lapply`-like loop: obtain work, do calculations, gather all results.

```
> lapplys <- function(X, FUN, ..., comm = 1) {
+     rank <- mpi.comm.rank(comm) + 1
+     n <- mpi.comm.size(comm)
+     tasks <- 1:length(X)
+     mywork <- X[split(tasks, cut(tasks, n))[[rank]]]
+     result <- lapply(mywork, FUN, ...)
+     allgather.Robj(result, comm)
+ }
```

Lab, part 2

Next: advanced topics, tools, and opportunities

The bootstrap

```
> library(boot)
> args(boot)
function (data, statistic, R, sim = "ordinary", stype = "i",
    strata = rep(1, n), L = NULL, m = 0, weights = NULL, ran.gen
        p) d, mle = NULL, ...)
NULL
```

- Bootstrap often embarassingly parallel, but...
- `boot` not readily accessible to parallelization.

One solution.

- Wrap `statistic` to distribute computing.
- WARNING: this is a 'hack', and will not work for parametric bootstraps.

Bootstrap first attempts

```
> ratio <- function(d, w) sum(d$x * w)/sum(d$u *
+     w)
> boot(city, ratio, R = 999, stype = "w")

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = city, statistic = ratio, R = 999, stype = "w")


Bootstrap Statistics :
    original     bias    std. error
t1* 1.520313 0.04600615   0.2272670
```

Parallel processing ideas that don't work.

```
> mpi.bcast.cmd(library(boot))
> mpi.bcast.Robj2slave(ratio)
> res1 <- mpi.remote.exec(boot(city, ratio, R = 999,
+     stype = "w"))
> sz <- mpi.comm.size()
> res2 <- mpi.parLapply(1:sz, function(i) boot(city,
+     ratio, R = 999/sz, stype = "w"))
```

- `res1`: identical bootstraps on each node!
- `res2`: 999 (ish) bootstraps, but need to be collated!

How `boot` works, and how to parallelize it

How `boot` works.

- `for` loop, calling `statistic` in a pre-determined order.
- For most versions of `boot`, each call to `statistic` has no influence on subsequent program execution.

How to parallelize it.

- Arrange for calls to `statistic` to be distributed, and...
- Manager always recieves results.
- Workers calculate result (expensive) and forwards to manager, but only occasionally.
- `wrap`: see the lab for details.

A wrapped `ratio`

Manager `ratiow`

```
> ratiow <- wrap(ratio, pseudo = 1)
> ratiow

function (...)
mpi.recv.Robj(mpi.any.source(), tag$result, comm)
<environment: 0x19e1a78>
```

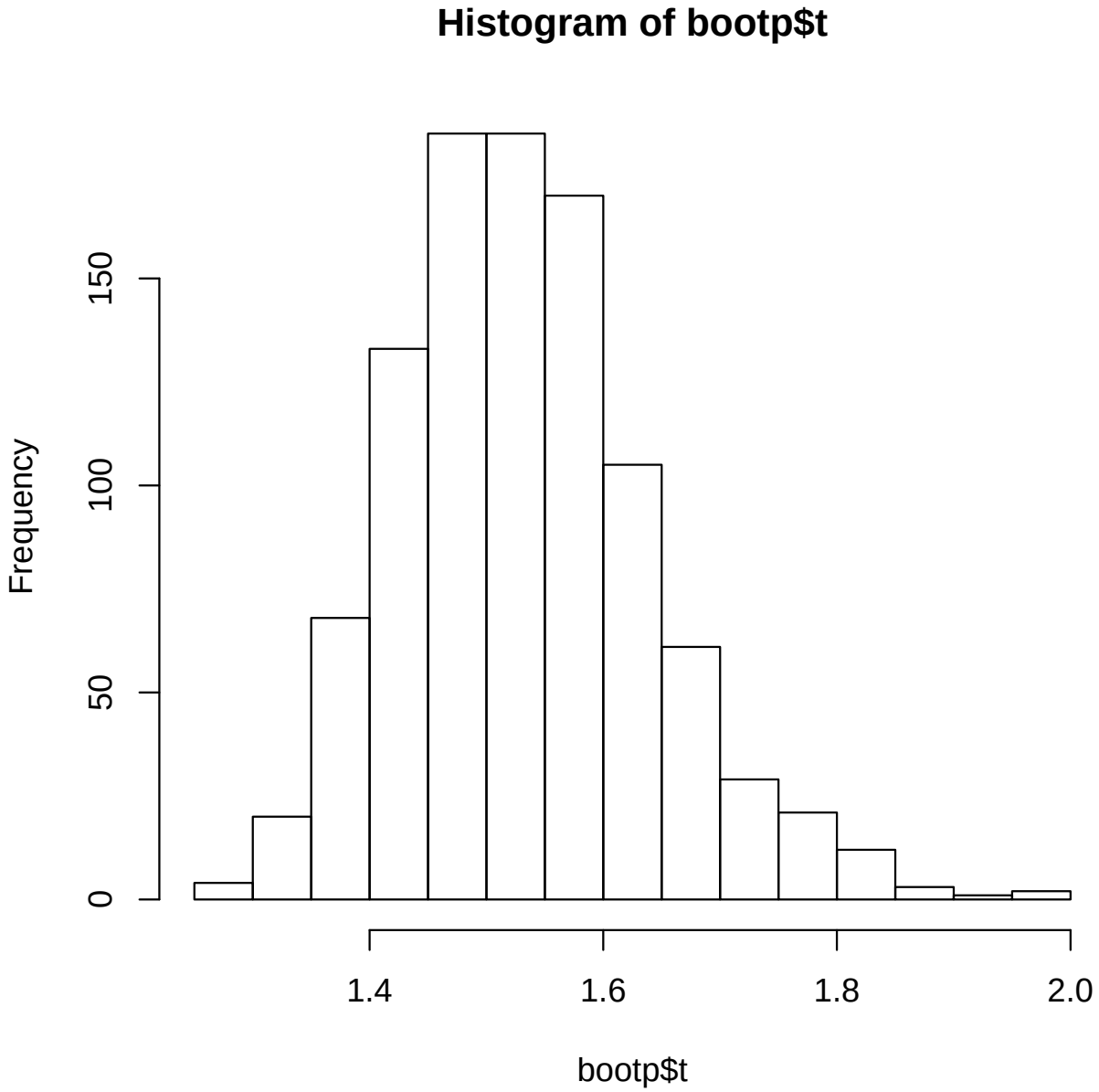
Worker `ratiow`

```
> mpi.bcast.Robj2slave(wrap)
> mpi.bcast.cmd(ratiow <- wrap(ratio, pseudo = 1))
> mpi.remote.exec(ratiow)[1]

$slave1
function (...)
{
```

```
    if (iter%%sz == rank - 1) {
        result <- func(...)
        mpi.send.Robj(result, 0, tag$result, comm)
    }
    else result <- pseudo
    iter <<- iter + 1
    result
}
<environment: 0xc65d40>

> mpi.bcast.cmd(boot(city, ratiow, R = 999, stype = "v"))
> bootp <- boot(city, ratiow, R = 999, stype = "v")
> hist(bootp$t)
```


Histogram of bootp$t
Frequency
0
50
100
150
1.4
1.6
1.8
2.0
bootp$t

Interfacing C code

R packages.

- Allow user to define collections of useful functions.
- Can include C source code, callable from R
- Package-writing details complex, but available online[a].

Strategy.

- Call a C function.
- Evaluate arbitrary parallel code, including spawning new processes.
- Return result.

[a] http://cran.fhcrc.org/doc/manuals/R-exts.html

A brief outline: spawning parallel processes

R code

```
> parallelPi <- function(nodes) {
+     nodes <- as.integer(nodes)
+     if (nodes < 1 || length(nodes) > 1)
+         stop("nodes should be a single integer value")
+     prog <- system.file(file.path("examples",
+         "cpi"), package = "Rpi")
+     .Call("parallelPi", prog, nodes, PACKAGE = "Rpi")
+ }
```

C function (to calculate π).

```
SEXP parallelPi(SEXP program, SEXP nodes) {
  MPI_Comm      intercomm;
  int           *slaverrcode;
  SEXP          ans;
```

```
  /* type checking & set-up */
  if(!isInteger(nodes) || (length(nodes) != 1))
    error("expected a numeric value for number of nodes");
  PROTECT(ans = allocVector(REALSXP, 1));
  /* parallel */
  slaverrcode = ( int* ) Calloc( nodes, int );
  MPI_Comm_spawn(CHAR(STRING_ELT(program, 0)), MPI_ARGV_NULL,
                 INTEGER(nodes)[0], MPI_INFO_NULL, 0,
                 MPI_COMM_SELF, &intercomm, slaverrcode);
  [...]
  UNPROTECT(1);
  return(ans);
}
```

Other opportunities: **Rmpi**-like packages

- rpvm provides an interface to the PVM library (similar to MPI); additional functionality not as developed as Rmpi.
- snow provides a common interface to point-to-point commands (like `mpi.send.Robj`) and functions (like `mpi.parLapply`) built on top of Rmpi, rpvm, or native 'sockets'.
- papply simple `mpi.parLapply` functionality, used transparently in serial or parallel modes.

NetWorkSpaces

- NetWorkSpaces[a] allows variables to be stored on a centralized server, and accessed by multiple instances of R – the illusion of shared memory.
- Multi-language (R, Python, matlab) support, i.e., possible to store a variable in matlab, access and manipulate it in R, and forward the result to Python.

[a] http://nws-r.sourceforge.net

Tools used today

Software infrastructure

- MPI (e.g., LAM/MPI[a]) for linux, MPICH2[b] for Windows.
- Intentionally clustered computers likely already have MPI.

Rmpi

- Linux: usually `biocLite(Rmpi)`.
- Windows: binary[c] download and instructions.

Quantian[d]

- Linux distribution available on a single bootable CD.
- Contains MPI, R, and most CRAN and Bioconductor packages!

[a] http://www.lam-mpi.org/
[b] http://www-unix.mcs.anl.gov/mpi/mpich/
[c] http://www.stats.uwo.ca/faculty/yu/Rmpi/
[d] http://dirk.eddelbuettel.com/quantian.html

Ideas for development

- Exposing existing functionality for parallelization, e.g., `xval` in MLInterfaces.
- Building high-level abstractions to MPI, e.g., automatically partitioning work in batch jobs, creatign the illusion of shared variables.
- Providing creative solutions to interactive parallel programming.