Simple Parallel Statistical Computing in R

Luke Tierney
Department of Statistics & Actuarial Science
University of Iowa
Joint work with A. J. Rossini and Na Li
Biostatistics, University of Washington

March 13, 2003

What Is R?

- R is a language for statistical computing and graphics.
- Similar to S (John Chambers et al., Bell Labs).
  - ACM Software System Award, 1999.
  - De facto standard for computing in Statistical research.
  - Documented in many books, e.g. Venables and Ripley.
- Can view R as a different implementation or dialect of S.

There are some important differences, but much code written for S runs unaltered under R.

History and Development Model

- R is an Open Source project.
- Originally developed by Robert Gentleman and Ross Ihaka.
- Developed by the R-core group since mid 1997,

Douglas Bates	John Chambers	Peter Dalgaard
Robert Gentleman	Kurt Hornik	Stefano Iacus
Ross Ihaka	Friedrich Leisch	Thomas Lumley
Martin Maechler	Guido Masarotto	Paul Murrell
Brian Ripley	Duncan Temple Lang	Luke Tierney

Why Parallel Computing?

- Many computations seem instantaneous.
- Some would take hours, days, or months.
- Often multiple processors are available:
  - multiple workstations
  - dedicated cluster
  - high-end SMP machine
- Can we make effective use of these resources?

Ideal Performance Improvement

- p processors should be p times faster than one processor.
- Some time scales:

Single processor	30 Processors
1 minute	2 seconds
1 hour	2 minutes
1 day	1 hour
1 month	1 day
1 year	2 weeks

Ideal Programming Requirement

- Minimal effort for simple problems.
- Be able to use existing high level (i.e. R) code.
- Ability to test code in sequential setting.

Parallel Computing on Networks of Workstations

- Use multiple cooperating processes.
- One process per available processor.
- Processes need to communicate with each other.
- Usually one process communicates with the user.

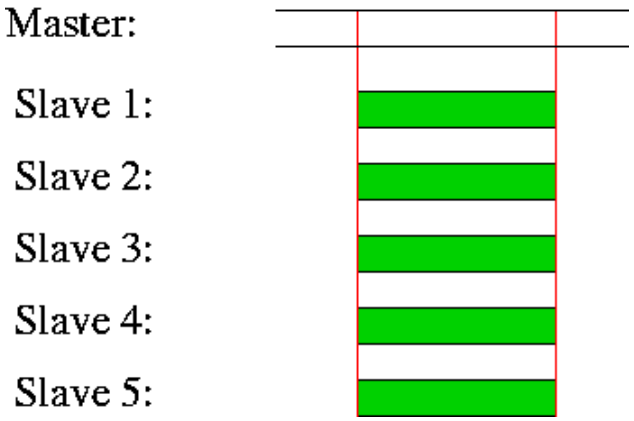
Available Communications Mechanisms

- Sockets
- Message passing libraries (PVM, MPI)
  - very powerful
  - not easy to use
  - designed for C, FORTRAN
- R interfaces
  - socket connections
  - rpvm, Rmpi

Master/Slave Model

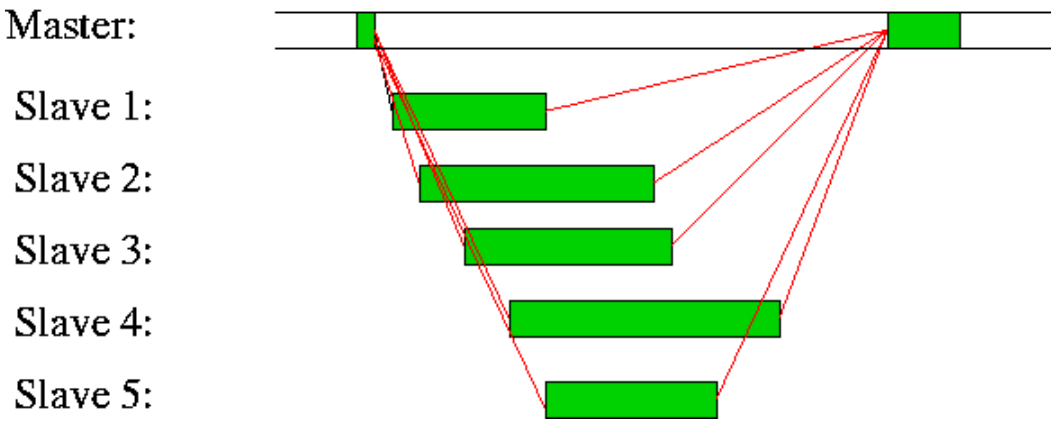
- Start with an "embarrassingly parallel" problem:

- Divide jobs among slave processes and collect results:



- Ideal: p times faster with p slaves.

A More Realistic Picture



- Jobs vary in complexity.
- Machines vary in speed/load.
- Communication takes time.
- Dividing up jobs and collecting results takes time.

SNOW: Simple Network of Workstations

- Snow is a package for R (available from CRAN).
- Snow uses the master/slave model.
  - The user starts an ordinary R session
  - The R session creates a cluster of slave processes.
  - Jobs are sent to the slave processes and results are returned.
- Communication can use sockets, PVM, MPI.

Starting A SNOW Cluster

- Create a cluster of 10 R slave processes:

```
library(snow)
cl <- makeCluster(10)
```

- Find out where the processes are running:

```
> do.call("rbind", clusterCall(cl, function(cl) Sys.info()["nodename"]))
      nodename
 [1,] "node02"
 [2,] "node03"
 ...
 [5,] "node06"
 [6,] "beowulf.stat.uiowa.edu"
 [7,] "node02"
 ...
[10,] "node05"
```

Stopping A SNOW Cluster

- Stop the cluster:

```
stopCluster(cl)
```

- Emergency break: Exit R, and
  - for PVM, halt the PVM.
  - for LAM-MPI, use `lamhalt` or `wipe`
  - for sockets, should just stop; if not, yoyo

Cluster Level Functions

- Call function on all nodes:

```
clusterCall(cl, exp, 1)
```

- Evaluate an expression on all nodes:

```
clusterEvalQ(cl, library(boot))
```

- Apply function to list, one element per node:

```
clusterApply(cl, 1:5, get("+"), 2)
```

Higher Level Functions

- Parallel `lapply`

```
> unlist(parLapply(cl, 1:15, get("+"), 2))
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

- Parallel `sapply`

```
> parSapply(cl, 1:15, get("+"), 2)
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

- Parallel `apply`

```
> parApply(cl, matrix(1:10, ncol=2), 2, sum)
[1] 15 40
```

Parallel Random Numbers

- Random number generation needs help:

```
> clusterCall(cl, runif, 3)
[[1]]
[1] 0.4351672 0.7394578 0.2008757
[[2]]
[1] 0.4351672 0.7394578 0.2008757
...
[[10]]
[1] 0.4351672 0.7394578 0.2008757
```

- Identical streams are likely, not guaranteed.

One Solution: SPRNG

- Scalable Parallel Random Number Generator library
- R interface `rsprng` (Na Li)
- Snow provides a convenience function:

```
> clusterSetupSPRNG(cl)
> clusterCall(cl, runif, 3)
[[1]]
[1] 0.014266542 0.749391854 0.007316102
[[2]]
[1] 0.8390032 0.8424790 0.8896625
...
[[10]]
[1] 0.591217470 0.121211511 0.002844222
```

Example: Parallel Bootstrap

- Bootstrapping is embarrassingly parallel.
- Replications can be split onto a cluster.
- Random number streams on nodes need to be independent.
- `boot` package allows bootstrapping of any R function.
- Help page shows example of bootstrapping `glm` fit for data on the cost of constructing nuclear power plants.

Example: Parallel Bootstrap (cont.)

- 1000 replicates on a single processor:

```
> wallTime(nuke.boot <-
+    boot(nuke.data, nuke.fun, R=1000, m=1,
+         fit.pred=new.fit, x.pred=new.data))
[1] 27.44
```

- Parallel version: 100 replicates on each of 10 cluster nodes:

```
> clusterSetupSPRNG(cl)
> clusterEvalQ(cl,library(boot))
> wallTime(cl.nuke.boot <-
+    clusterCall(cl,boot,nuke.data, nuke.fun, R=100, m=1,
+                fit.pred=new.fit, x.pred=new.data))
[1] 3.03
```

Example: Parallel Kriging

- Several R packages provide spatial prediction (kriging).
- Sgeostat has a pure R version, `krige`.
- Computation is a simple loop over points.
- Fairly slow when using only points within `maxdist`.
- Result structure is fairly simple.
- Easy to write a parallel version.

Parallel Version of `krige`

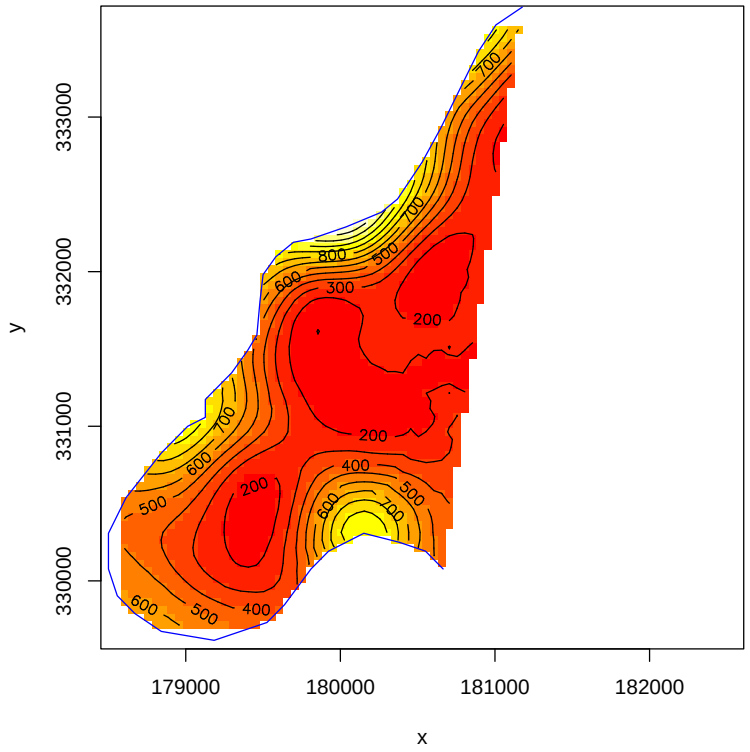
```
parKrige <- function(cl, s, ...) {
    # split the prediction points s
    idx <- clusterSplit(cl, 1: dim(s)[1])
    ssplt <- lapply(idx, function(i) s[i,])

    # compute the predictions in parallel
    v <- clusterApply(cl, ssplt, krige, ...)

    # assemble and return the results
    merge <- function(x, f) do.call("c", lapply(x, f))
    s.o <- point(s)
    s.o$zhat <- merge(v, function(y) y$zhat)
    s.o$sigma2hat <- merge(v, function(y) y$sigma2hat)
    return(s.o)
}
```

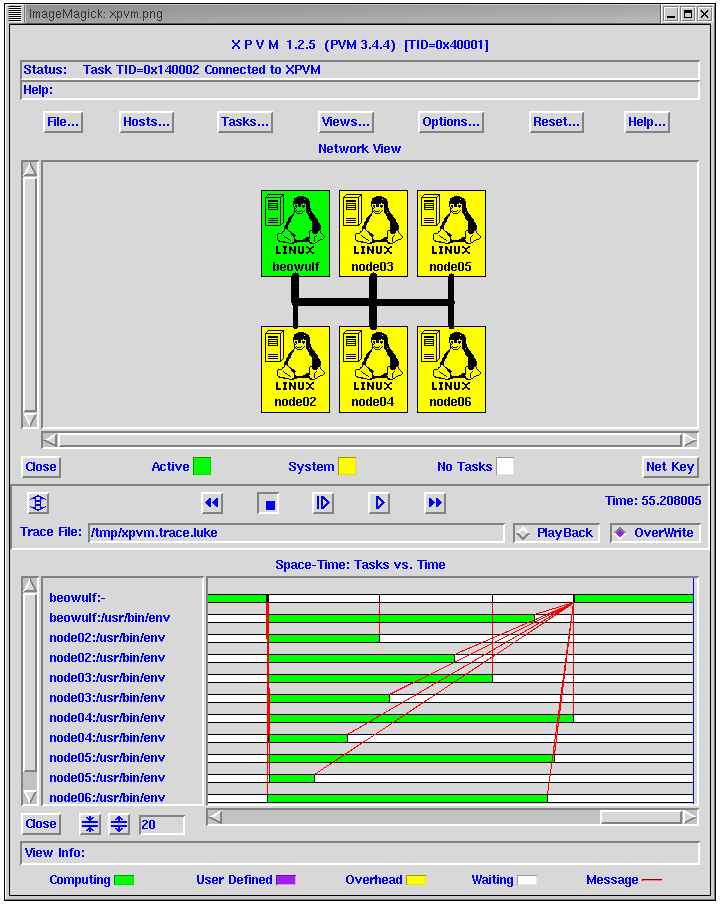
Zink in Maas Flood Plane Ground Water

- Measurements at 155 points.
- Predict on $50m \times 50m$ grid.
- Use only data within 1 kilometer.
- Sequential version takes 38.12 seconds.
- Parallel version (10 nodes) takes 6.22 seconds.
- Only a factor of 6 speedup.



XPVM: Visualizing the Parallel Computation

- Graphical console for starting/stopping PVM.
- Shows hosts used in the PVM (all dual processor).
- Displays activity dynamically.
- Shows uneven load distribution.



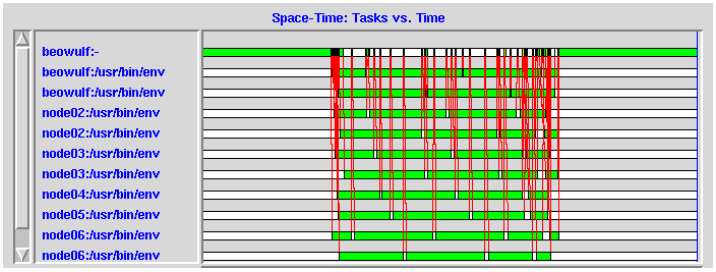
Load Balanced Kriging

- `clusterApplyLB`: load balanced `clusterApply`.
- Give more jobs n than cluster nodes p.
  - Places first p jobs on p nodes,
  - job $p+1$ goes to first node to finish,
  - job $p+2$ goes to second node to finish,
  - etc., until all n jobs are done.

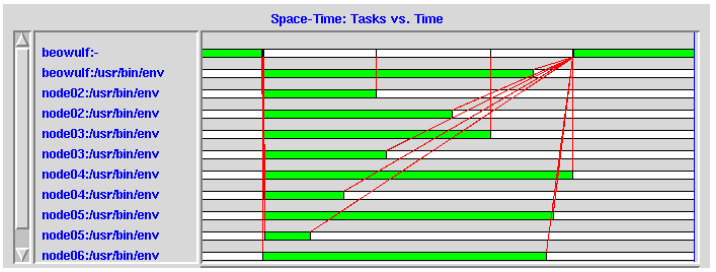
Load Balanced Kriging (cont.)

- Load balanced version takes 4.62 seconds (speedup of 8.25).

Load Balanced



No Load Balancing



- Communication is increased.
- Node executing a particular job is non-deterministic.

Example: Cross Validation

- Useful for choosing tuning parameters.
- Common structure:
  - Outer loop over tuning parameters
  - Inner loop over omitted data
  - Additional inner replication loop if random (`nnet`)
- Good initial approach:
  - parallelize loop over omitted data
  - replace loop by `lapply`; test and debug
  - replace `lapply` by `parLapply`

Example: Cross Validation (cont.)

Nested loops

```
cv <- function(parameters, data)
    for (p in parameters) {
        v <- vector("list", length(data))
        for (d in data)
            v[[d]] <- fit for p, omitting d
        summarize result for p
    }
```

lapply in inner loop

```
lcv <- function(parameters, data)
    for (p in parameters) {
        fit <- function(p, d)
            fit for p, omitting d
        v <- lapply(data, fit)
        summarize result for p
    }
```

Parallel version

```
parCv <- function(cl, parameters, data)
    for (p in parameters) {
        fit <- function(p, d)
            fit for p, omitting d
        v <- parLapply(cl, data, fit)
        summarize result for p
    }
```

Different Example: Parallelized Animation

- Animation shows effect of varying one parameter.
- Needs several frames per second to be effective.
- Sometimes frames take several seconds to compute.
- Parallel frame computation can help.

Performance Consideration

- Communication.
  - explicit data
  - hidden data
- Load balancing.
  - variable task complexities
  - variable node performance/load

Discussion

- Design goals:
  - simplicity of design
  - portable
  - easy to use
  - user cannot deadlock
- Drawbacks:
  - cannot express all parallel algorithms
  - some can be expressed but not efficiently

Future Directions

- Issues to address:
  - better error handling
  - sensible handling of user interrupts
- Extensions
  - effective interface to queue/stream of jobs
  - parallel animation tools
  - inter-node communication (BSP?)