More snow Examples

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

September 27, 2007

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

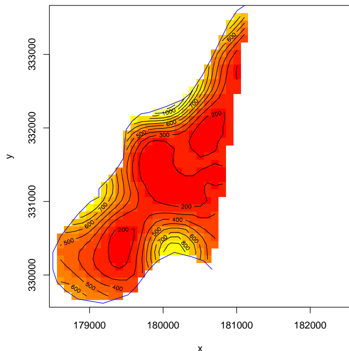
```
parKrige <- function(cl, s, ...) {
    # split the prediction points s
    idx <- clusterSplit(cl, 1: dim(s)[1])
    ssplt <- lapply(idx, function(i) s[i,])

    # 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$do <- merge(v, function(y) y$do)
    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 2.53 seconds.
- Parallel version (10 nodes) takes 0.38 seconds.
- Only a factor of 6.7 speedup.



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.

Parallel Version of krige with Load Balancing

```
parKrigeLB <- function(cl, s, ..., LBF = 4) {
    # split the prediction points s
    idx <- splitIndices(dim(s)[1], length(cl) * LBF) #****
    ssplt <- lapply(idx, function(i) s[i,])

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

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

Results and Notes

- The result is a speedup of 8.5:

```
> system.time(pklb <-
+    parKrigeLB(cl,grid$point,maas.point,'zinc',maas.vmod,
+               maxdist=1000,extrap=FALSE,border=maas.bank))
   user  system elapsed
  0.108   0.007   0.297
```

- And the results are identical to the sequential version.

```
> identical(k, pk)
[1] TRUE
> identical(pklb, pk)
[1] TRUE
```

- Load balancing can improve performance.
- Load balancing does increase communication somewhat.
- Load balancing is hard to combine with reproducible simulation.
- Load balancing can be hard to use with distributed data.

Parallel 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`

Parallel Cross Validation

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

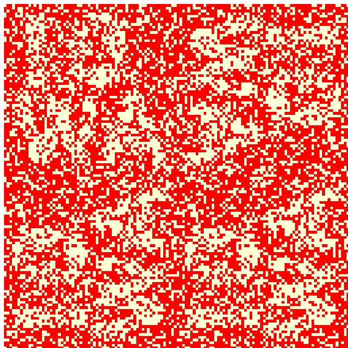
```
parCv <- function(cl, parameters, data)
    for (p in parameters) {
        fit <- function(p, d)
            fit for p, omitting d
        v <- parLapply(cl, data, fit)
        summarize result for p
    }
```

Simple Image Reconstruction

- Observe y_{ij}, a binary image with noise.
- True image is x_{ij}.
- Simple noise model:
  - $y_{ij}|x \sim p^{y_{ij}=x_{ij}}(1-p)^{y_{ij}\neq x_{ij}}$
  - pixel noise is independent
- A simple image prior distribution:

$$p(x) \propto \exp\{\beta N(x)\}$$

  with $N(x)$ the number of neighbor pairs that are the same color.
- This is the *Ising model*.
- Simplest approach uses 4 neighbors.
- Dependence increases with β.

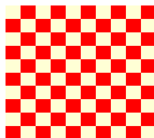


Simple Image Reconstruction

- Full conditionals are simple:

$$p(x_{ij} = c|y, \{x \setminus x_{ij}\}) = \exp\{\beta \times \text{number of neighbors with color } c\}$$

- Gibbs sampling is easy but can be slow in R using loops.
- Can vectorize using checkerboard order:



- Given the red pixels, the white pixels are independent.
- This also allows parallel computation.

A Vectorized Algorithm

Computing the number of neighbors with a specified color:

```
nn <- function(m, c) {
    nr <- nrow(m)
    nc <- ncol(m)
    nn <- matrix(0, nr, nc)
    nn[1:(nr)-1,] <- nn[1:(nr)-1,] + (m[2:nr,] == c)
    nn[2:nr,] <- nn[2:nr,] + (m[1:(nr-1),] == c)
    nn[,1:(nc)-1] <- nn[,1:(nc)-1] + (m[,2:nc] == c)
    nn[,2:nc] <- nn[,2:nc] + (m[,1:(nc-1)] == c)
    nn
}
```

Simulating new pixel values:

```
simGroup <- function(m, l2, l1, beta, which) {
    pp2 <- l2 * exp(beta * nn(m, 2))
    pp1 <- l1 * exp(beta * nn(m, 1))
    pp <- pp2 / (pp2 + pp1)
    m[which] <- ifelse(runif(sum(which)) < pp[which], 2, 1)
    m
}
```

A Vectorized Algorithm

Computing the likelihood:

```
makeLik <- function(img, p)
    list(ifelse(img == 1, p, 1 - p), ifelse(img == 2, p, 1 - p))
```

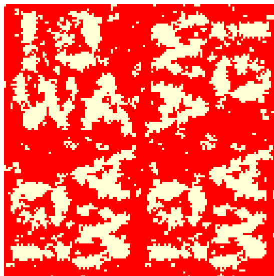
Estimating the posterior mean image:

```
simImgPM <- function(img, beta, p, NN = 50, K = 50) {
    lik <- makeLik(img, p)
    white <- outer(1:nrow(img), 1:ncol(img), FUN=`+`) %% 2 == 1
    black <- ! white
    m <- img
    mm <- 0
    for (N in 0 : (NN - 1)) {
        for (i in 1 : K) {
            m <- simGroup(m, lik[[2]], lik[[1]], beta, white)
            m <- simGroup(m, lik[[2]], lik[[1]], beta, black)
        }
        mm <- (N / (N + 1)) * mm + (1 / (N + 1)) * m
    }
    mm
}
```

A Vectorized Algorithm

Some results for $NN = 500$, $K = 1$, and $p = 0.7$:

$\beta = 0.7$



$\beta = 0.9$

Timing:

```
> system.time(pm <- simImgPM(img4, 0.9,0.7,500,1))
   user  system elapsed
 42.914   0.468  43.418
```

Towards a Simple Parallel Version

- Split image into two groups of columns.
- Each one needs edge of the other group.
- Find split point and label likelihood values:

```
n2 <- floor(ncol(img) / 2)
i1 <- 1 : n2; i1x <- c(i1, n2 + 1)
i2 <- (n2 + 1) : ncol(img); i2x <- c(n2, i2)
l1 <- lik[[1]]
l2 <- lik[[2]]
```

- Simulate "white" pattern in two steps

```
m[,i1] <- simGroup(m[,i1x], l2[,i1x], l1[,i1x], beta,
                   white[,i1x])[,-(n2 + 1)]
m[,i2] <- simGroup(m[,i2x], l2[,i2x], l1[,i2x], beta,
                   white[,i2x])[,-1]
```

- Do the same for "black" pattern.

A Simple Parallel Version

- Replace the “white” step by

```
v <- clusterMap(cl, fun,
                list(m[,i1x], m[,i2x]),
                list(l2[,i1x], l2[,i2x]),
                list(l1[,i1x], l1[,i2x]),
                list(white[,i1x], white[,i2x]))
m[,i1] <- v[[1]][,-(n2 + 1)]
m[,i2] <- v[[2]][,-1]
```

  with `fun` created by

```
makeSimImgHelper <- function(beta)
    function(m, l2, l1, which)
        simGroup(m, l2, l1, beta, which)
```

  Need to export `nn` and `simGroup` to nodes.

- Timing: 104 seconds—almost 3 times *slower*.
- Problem: too much comunication.
- Solution: maintain state on the nodes.

Maintaining Sampler State

```
makeImgSampler <- function(img, beta, p, white.first = FALSE) {
    m<-img
    mm<-img
    <<more setup code>>
    stepWhite <- function() {
        m <<- simGroup(m, l2, l1, beta, white)
        invisible(NULL)
    }
    stepBlack <- function() ...
    accumulate <- function() ...
    getColumn <- function(i) m[,i]
    setColumn <- function(i, v) m[,i] <<- v
    pm <- function() mm
    list(stepWhite = stepWhite, stepBlack = stepBlack,
         accumulate = accumulate,
         getColumn = getColumn, setColumn = setColumn, pm = pm)
}
```

Serial Algorithm Using Sampler State

- The code:

```
simImgPM <- function(img, beta, p, NN = 50, K = 50) {
    smp <- makeImgSampler(img, beta, p)
    for (i in 1:NN) {
        for (i in 1:K) {
            smp$stepWhite()
            smp$stepBlack()
        }
        smp$accumulate()
    }
    smp$pm()
}
```

- This is an *object oriented* design.
- `smp` is a sampler object with mutable state.
- The algorithem proceeds by sending messages to `smp`.

Towards a Parallel Version

- Create two objects and get the border columns:

```
wf2 <- if (nc2 %% 2 == 0) FALSE else TRUE
smp1 <- makeImgSampler(img[,1:(nc2+1)], beta, p)
smp2 <- makeImgSampler(img[,nc2:nc], beta, p, wf2)
c1 <- smp1$getColumn(nc2)
c2 <- smp2$getColumn(2)
```

- "White" step:

```
v <- list(stepWhite2(smp1, nc2 + 1, c2, nc2),
          stepWhite2(smp2, 1, c1, 2))
c1 <- v[[1]]
c2 <- v[[2]]
```

with

```
stepWhite2 <- function(smp, i, v, j) {
    smp$setColumn(i, v)
    smp$stepWhite()
    smp$getColumn(j)
}
```

Parallel Version

- Sampler will be in a global variable `smp` on each node:

```
nodePutSmp <- function(v) {
    assign("smp", v, envir = .GlobalEnv);
    NULL
}
clusterApply(cl, list(smp1, smp2), nodePutSmp)
```

  Some utility functions that use this global value:

```
nodeStepWhite2 <- function(i, v, j) {
    stepWhite2(smp, i, v, j);
    NULL
}
nodeStepBlack2 <- function(i, v, j) ...
nodeAccumulate <- function() { smp$accumulate(); NULL }
nodePM <- function() smp$pm()
```

- Need to export `stepWhite2` and `stepBlack2` to nodes:

Parallel Version

- Body of the parallel code:

```
for (i in 1:NN) {
    for (j in 1:K) {
        v <- clusterMap(cl, nodeStepWhite2,
                        c(nc2 + 1, 1), list(c2, c1), c(nc2, 2))
        c1 <- v[[1]]
        c2 <- v[[2]]
        <<same for black pixels>>
    }
    clusterCall(cl, nodeAccumulate)
}
v <- clusterCall(cl, nodePM)
cbind(v[[1]][,-(nc2+1)], v[[2]][,-1])
```

- Timings for $NN = 500$, $K = 1$:

Serial version:	31.57 seconds
Parallel version:	17.56 seconds

Some Open Issues

- Fault tolerance:
  - nodes/communication can fail
  - R processes can crash
- Error handling
  - Some computations may result in R-level errors.
  - Currently these are returned as results of class `try-error`.
- Interrupt handling
- Load balancing issues:
  - load balancing and reproducible simulations
  - integrating load balancing with `clusterApply`, `parLapply`, etc..
  - convenient control options for load balancing (e.g. chunk size)
- Support for
  - intermediate communication between nodes
  - maintaining state on nodes

  The BSP model may be useful to explore.