Some snow Examples

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

September 20, 2007

Simple Examples

- 10 second sleep on each node:

```
> system.time(clusterCall(cl, Sys.sleep, 10))
   user  system elapsed
  0.025   0.001  10.019
```

- Parallel `qtukey` evaluation:

```
> x<-1:100/101
> system.time(qtukey(x, 2, df=2))
   user  system elapsed
  3.661   0.000   3.662
> system.time(parLapply(cl, x, qtukey, 2, df=2))
   user  system elapsed
  0.007   0.000   0.436
```

  Timings are uneven across nodes.

Simple Matrix Multiply

- The current `parMM` computes AB on p nodes by
  - partitioning a into p blocks of rows $A_1, \ldots, A_p$
  - computing $A_i B$ on note i
- This sends one block of A and all of B to each node.
- Communication increases linearly in p.
- Using too many nodes will slow the computation down.

An Alternative Algorithm

- Partition
  - A into a blocks of rows $A_1, \ldots, A_a$.
  - B into b blocks of columns $B_1, \ldots, B_b$.

  with $ab \leq p$.
- Then

$$AB = \begin{bmatrix} A_1 \\ \vdots \\ A_a \end{bmatrix} \begin{bmatrix} B_1 & \cdots & B_b \end{bmatrix} = \begin{bmatrix} A_1 B_1 & \cdots & A_1 B_b \\ \vdots & \ddots & \vdots \\ A_a B_1 & \cdots & A_a B_b \end{bmatrix}$$

- If A, B are $n \times n$ and $a = b = \sqrt{p}$ then the number of elements sent to nodes is

$$2p \times n \times n/\sqrt{p} = 2n^2\sqrt{p}$$

- So communication increases as $\sqrt{p}$

An Alternative Algorithm

A first implementation:

```
parMM <- function(cl, A, B, Pmax = length(cl)) {
    r <- nrow(A)
    c <- ncol(B)
    pr <- min(Pmax, max(1, floor(sqrt(Pmax * r / c))))
    pc <- floor(Pmax / pr)
    A_rows <- rep(splitRows(A, pr), pc)
    B_cols <- rep(splitCols(B, pc), each = pr)
    args <- mapply(list, A_rows, B_cols, SIMPLIFY = FALSE)
    Cblocks <- clusterApply(cl, args,
                            function(arg) arg[[1]] %*% arg[[2]])
    C_cols <- lapply(splitList(Cblocks, pc),
                     function(x) docall(rbind, x))
    docall(cbind, C_cols)
}
```

An Alternative Algorithm

A modification that avoids variable capture:

```
parMMhelper <- function(arg) arg[[1]] %*% arg[[2]]

parMM <- function(cl, A, B, Pmax = length(cl)) {
    r <- nrow(A)
    c <- ncol(B)
    pr <- min(Pmax, max(1, floor(sqrt(Pmax * r / c))))
    pc <- floor(Pmax / pr)
    A_rows <- rep(splitRows(A, pr), pc)
    B_cols <- rep(splitCols(B, pc), each = pr)
    args <- mapply(list, A_rows, B_cols, SIMPLIFY = FALSE)
    Cblocks <- clusterApply(cl, args, parMMhelper)
    C_cols <- lapply(splitList(Cblocks, pc),
                     function(x) docall(rbind, x))
    docall(cbind, C_cols)
}
```

An Alternative Algorithm

Using the new `clusterMap` function:

```
parMM <- function(cl, A, B, Pmax = length(cl)) {
    r <- nrow(A)
    c <- ncol(B)
    pr <- min(Pmax, max(1, floor(sqrt(Pmax * r / c))))
    pc <- floor(Pmax / pr)
    A_rows <- rep(splitRows(A, pr), pc)
    B_cols <- rep(splitCols(B, pc), each = pr)
    Cblocks <- clusterMap(cl, `%*%`, A_rows, B_cols)
    C_cols <- lapply(splitList(Cblocks, pc),
                     function(x) docall(rbind, x))
    docall(cbind, C_cols)
}
```

Two Classical Algorithms

- Fox, Cannon algorithms for computing $C = AB$.
- Both algorithms
  - partition C into blocks
  - partition the row/column blocks of A, B into blocks
  - start with one block of A, one of B, and one of C on each node
  - multiply their A, B blocks and place in C
  - get new A, B blocks, multiply, add into C block
  - repeat until done
- The algorithms differ in how A, B blocks are moved.
- At any time each node contains only three blocks
- All three matrices can be too large to fit in memory on any one machine.
- Communication can run in parallel.

Parallel Bootstrap

- Classical example of an *embarrassingly parallel* computation.
- Need to be careful about random numbers.
- Serial version of an example from the boot help page:

```
> R <- 1000
> system.time(nuke.boot <-
+             boot(nuke.data, nuke.fun, R=R, m=1,
+                  fit.pred=new.fit, x.pred=new.data))
   user  system elapsed
 12.703   0.001  12.706
```

- Parallel version, using 10 nodes:

```
> clusterEvalQ(cl,library(boot))
> clusterSetupRNG(cl)
> system.time(cl.nuke.boot <-
+             clusterCall(cl,boot,nuke.data, nuke.fun,
+                         R=R/length(cl), m=1,
+                         fit.pred=new.fit, x.pred=new.data))
   user  system elapsed
  0.009   0.004   1.246
```

Parallel Bootstrap

- To be useful, we need to merge the list of results.
- A simple merging function:

```
fixboot <- function(bootlist) {
    boot <- bootlist[[1]]
    boot$t <- do.call(rbind,lapply(bootlist, function(x) x$t))
    boot$R <- sum(sapply(bootlist, function(x) x$R))
    if (! is.null(boot$pred.i))
        boot$pred.i <- do.call(rbind,lapply(bootlist,
                                            function(x) x$pred.i))
    boot
}
```

Parallel Bootstrap

- Fixing up the result:

```
cl.nuke.boot.fixed <- fixboot(cl.nuke.boot)
```

- Bootstrap prediction errors:

```
> mean(nuke.boot$t[,8]^2)
[1] 0.08511571
> mean(cl.nuke.boot.fixed$t[,8]^2)
[1] 0.09392631
```

- Basic bootstrap prediction limits:

```
> new.fit-sort(nuke.boot$t[,8])[c(975,25)]
[1] 6.098594 7.263207
> new.fit-sort(cl.nuke.boot.fixed$t[,8])[c(975,25)]
[1] 6.137178 7.304385
```