deathstar – seamless cloud computing for R

Whit Armstrong
armstrong.whit@gmail.com

KLS Diversified Asset Management

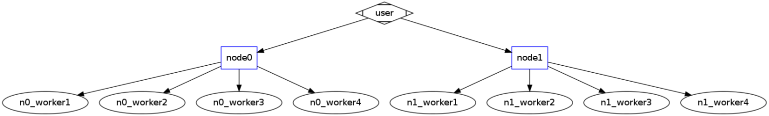
May 11, 2012

What is deathstar?

- deathstar is a dead simple way to run lapply across EC2 nodes or local compute nodes
- deathstar uses ZMQ (via the rzmq package) to deliver both data and code over the wire
- deathstar consists of two components: a daemon and an R package
- simplicity is the key theme

deathstar – visual

- user – user workstation
- node0,node1 – local or cloud machines
- n0_workers, n1_workers – deathstar workers



deathstar daemon – details

- deathstar is a daemon that listens on two ports
- the 'exec' port listens for incoming work
- the 'status' port fulfills queries on the current workload capacity of the server (this is what allows dynamic job allocation)

deathstar – queue device (about 100 lines of c++)

```
void deathstar(zmq::socket_t& frontend, zmq::socket_t& backend, zmq::socket_t& worker_signal, zmq::socket_t& status) {
  int number_of_workers(0);

  // loosely copied from lazy pirate (ZMQ guide)
  while (1) {
    zmq_pollitem_t items [] = {
      { worker_signal, 0, ZMQ_POLLIN, 0 },
      { status, 0, ZMQ_POLLIN, 0 },
      { backend,  0, ZMQ_POLLIN, 0 },
      { frontend, 0, ZMQ_POLLIN, 0 }
    };
    // frontend only if we have available workers
    // otherwise poll both worker_signal and backend
    int rc = zmq_poll(items, number_of_workers ? 4 : 3, -1);

    //  Interrupted
    if(rc == -1) {
      break;
    }

    //  worker_signal -- worker came online
    if(items[0].revents & ZMQ_POLLIN) {
      process_worker_signal(worker_signal, number_of_workers);
    }

    //  status -- client status request
    if(items[1].revents & ZMQ_POLLIN) {
      process_status_request(status, number_of_workers);
    }

    //  backend -- send data from worker to client
    if(items[2].revents & ZMQ_POLLIN) {
      process_backend(frontend, backend);
    }

    // frontend -- send data from client to worker
    if (items[3].revents & ZMQ_POLLIN) {
      process_frontend(frontend, backend, number_of_workers);
    }
  }
}
```

deathstar – worker (25 SLOC of R)

```
#!/usr/bin/env Rscript

library(rzmq)

worker.id <- paste(Sys.info()["nodename"],Sys.getpid(),sep=":")

cmd.args <- commandArgs(trailingOnly=TRUE)

print(cmd.args)

work.endpoint <- cmd.args[1]
ready.endpoint <- cmd.args[2]
log.file <- cmd.args[3]

sink(log.file)

context = init.context()

ready.socket = init.socket(context,"ZMQ_PUSH")
work.socket = init.socket(context,"ZMQ_REP")

connect.socket(ready.socket,ready.endpoint)
connect.socket(work.socket,work.endpoint)

while(1) {
    ## send control message to indicate worker is up
    send.null.msg(ready.socket)

    ## wait for work
    msg = receive.socket(work.socket);

    index <- msg$index
    fun <- msg$fun
    args <- msg$args
    print(system.time(result <- try(do.call(fun,args),silent=TRUE)))
    send.socket(work.socket,list(index=index,result=result,node=worker.id));
    print(gc(verbose=TRUE))
}
```

deathstar package – exposes zmq.cluster.lapply

code is a little too long for a slide, but does ~~three~~two basic things:

- check node capacity, if *capacity* > 0, submit job
- collect jobs
- node life cycle monitoring (future feature)

So, what can it do?

Stochastic estimation of Pi

Borrowed from JD Long:
http://www.vcasmo.com/video/drewconway/8468
or
http://joefreeman.co.uk/blog/2009/07/estimating-pi-with-monte-carlo-methods

```r
estimatePi <- function(seed,draws) {
    set.seed(seed)
    r <- .5
    x <- runif(draws, min=-r, max=r)
    y <- runif(draws, min=-r, max=r)
    inCircle <- ifelse( (x^2 + y^2)^.5 < r , 1, 0)
    sum(inCircle) / length(inCircle) * 4
}
```

Basic lapply example – estimatePi

```
require(deathstar)
cluster <- c("krypton","xenon","node1","mongodb","research")
pi.local.runtime <- system.time(pi.local <-
zmq.cluster.lapply(cluster=cluster,1:1e4,estimatePi,draws=1e5))

print(mean(unlist(pi.local)))

## [1] 3.142

print(pi.local.runtime)

##    user  system elapsed
##   7.828   1.024  48.378

print(node.summary(attr(pi.local,"execution.report")))

##       node jobs.completed
## 1  krypton           2206
## 2  mongodb           2191
## 3    node1           1598
## 4 research           1357
## 5    xenon           2648
```

Simple beta calculation

```
calc.beta <- function(ticker,mkt,n.years) {
    require(KLS)
    x <- get.bbg(ticker); x <- tail(x,250*n.years)
    ticker <- gsub(" ",".",ticker)
    x.diff <- diff(x,1) * 100
    reg.data <- as.data.frame(cbind(x.diff,mkt))
    colnames(reg.data) <- c("asofdate",ticker,"SPX")
    formula <- paste(ticker,"SPX",sep="~")
    coef(lm(formula,data=reg.data))
}
```

Simple beta calculation – setup

```r
require(KLS)
spx <- get.bbg("SPXT Index")
spx.ret <- (spx/lag(spx, 1) - 1) * 100

tickers <- c("USGG30YR Index", "USGG10YR Index", "USGG5YR Index",
    "USGG2YR Index", "USGG3YR Index", "USSW10 Curncy", "USSW5 Curncy",
    "USSW2 Curncy", "TY1 Comdty", "CN1 Comdty", "RX1 Comdty", "G 1 Comdty",
    "JB1 Comdty", "XM1 Comdty")

cluster <- c("krypton", "research", "mongodb")
```

Simple beta calculation – results

```r
require(deathstar)
beta <- zmq.cluster.lapply(cluster = cluster, tickers, calc.beta,
    mkt = spx.ret, n.years = 4)
beta <- do.call(rbind, beta)
rownames(beta) <- tickers
print(beta)

##                (Intercept)      SPX
## USGG30YR Index     -0.2011   1.9487
## USGG10YR Index     -0.2695   2.0328
## USGG5YR Index      -0.3130   1.9442
## USGG2YR Index      -0.2738   1.3025
## USGG3YR Index      -0.1675   1.3840
## USSW10 Curncy      -0.3034   1.8889
## USSW5 Curncy       -0.3526   1.6594
## USSW2 Curncy       -0.3275   0.8852
## TY1 Comdty          3.0219 -13.1236
## CN1 Comdty          3.0726 -12.8818
## RX1 Comdty          3.9561 -11.6475
## G 1 Comdty          4.4650 -10.2193
## JB1 Comdty          1.0143  -0.8301
## XM1 Comdty          0.3558  -0.4453
```

So, what have we learned?

- deathstar looks pretty much like lapply, or parLapply
- it's pretty fast
- not much configuration

into the cloud!

Basic steps to customize your EC2 instance.
The deathstar daemon can be downloaded here:
`http://github.com/downloads/armstrtw/deathstar.core/deathstar_0.0.1-1_amd64.deb`

```
## launch EC2 instance from web console
## assume instance public dns is: ec2-23-20-55-67.compute-1.amazonaws.com

## from local workstation
scp rzmq_0.6.6.tar.gz \
ec2-23-20-55-67.compute-1.amazonaws.com:/home/ubuntu

scp ~/.s3cfg \
ec2-23-20-55-67.compute-1.amazonaws.com:/var/lib/deathstar

## from EC2 node
sudo apt-get install r-base-core r-base-dev libzmq-dev s3cmd
sudo R CMD INSTALL rzmq_0.6.6.tar.gz
sudo R CMD INSTALL AWS.tools_0.0.6.tar.gz
sudo dpkg -i deathstar_0.0.1-1_amd64.deb


## from local workstation
## check that the right ports are open (6000,6001)
## and AWS firewall is configured properly
nmap ec2-23-20-55-67.compute-1.amazonaws.com
```

a little more aggressive – estimatePi on EC2

c1.xlarge is an 8 core machine with 7 GB of RAM.

```
require(deathstar)
require(AWS.tools)

## start the EC2 cluster
cluster <- startCluster(ami = "ami-1c05a075", key = "kls-ec2",
    instance.count = 4, instance.type = "c1.xlarge")
nodes <- get.nodes(cluster)
pi.ec2.runtime <- system.time(pi.ec2 <- zmq.cluster.lapply(cluster = nodes,
    1:10000, estimatePi, draws = 1e+05))
```

estimatePi on EC2 – workload distribution

```r
print(mean(unlist(pi.ec2)))

## [1] 3.142

print(pi.ec2.runtime)

##    user  system elapsed
##   6.285   1.044 122.751

print(node.summary(attr(pi.ec2, "execution.report")))

##                node jobs.completed
## 1  ip-10-115-54-77           2510
## 2 ip-10-34-239-135           2526
## 3   ip-10-39-10-26           2521
## 4  ip-10-78-49-250           2443
```

mixed mode – use local + cloud resources

```
## the EC2 cluster is still running from the last example
cloud.nodes <- get.nodes(cluster)
nodes <- c(c("krypton", "xenon", "node1", "mongodb", "research"),
    cloud.nodes)
pi.mixed.runtime <- system.time(pi.mixed <- zmq.cluster.lapply(cluster = nodes,
    1:10000, estimatePi, draws = 1e+05))

## turn the cluster off
terminateCluster(cluster)

##      [,1]       [,2]         [,3]      [,4]
## [1,] "INSTANCE" "i-d35d82b5" "running" "shutting-down"
## [2,] "INSTANCE" "i-d15d82b7" "running" "shutting-down"
## [3,] "INSTANCE" "i-d75d82b1" "running" "shutting-down"
## [4,] "INSTANCE" "i-d55d82b3" "running" "shutting-down"
```

workload distribution for local + cloud

```
print(mean(unlist(pi.mixed)))

## [1] 3.142

print(pi.mixed.runtime)

##    user  system elapsed
##   5.812   0.708  36.957

print(node.summary(attr(pi.mixed, "execution.report")))

##               node jobs.completed
## 1  ip-10-115-54-77            815
## 2 ip-10-34-239-135            810
## 3   ip-10-39-10-26            826
## 4  ip-10-78-49-250            827
## 5          krypton           1470
## 6          mongodb           1648
## 7            node1           1051
## 8         research            940
## 9            xenon           1613
```

distributed memory regression

Borrowing from Thomas Lumley's presentation[1], one can calculate regression coefficients in a distributed memory environment via these steps:

- Compute $X^T X$ and $X^T y$ in chunks
- aggregate chunks
- reduce by $\hat{\beta} = (X^T X)^{-1} X^T y$

[1] `http://faculty.washington.edu/tlumley/tutorials/user-biglm.pdf`

distributed memory regression – data

Airline on-time performance
`http://stat-computing.org/dataexpo/2009/`

The data:
The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. This is a large dataset: there are nearly 120 million records in total, and takes up 1.6 gigabytes of space compressed and 12 gigabytes when uncompressed. To make sure that you're not overwhelmed by the size of the data, we've provide two brief introductions to some useful tools: linux command line tools and sqlite, a simple sql database.

distributed memory regression – setup

```
distributed.reg <- function(bucket, nodes, y.transform, x.transform) {
    calc.xTx.xTy <- function(chnk.name, y.transform, x.transform) {
        require(AWS.tools)
        raw.data <- s3.get(chnk.name)
        y <- y.transform(raw.data)
        X <- x.transform(raw.data)

        bad.mask <- is.na(y) | apply(is.na(X), 1, any)
        y <- y[!bad.mask]
        X <- X[!bad.mask, ]

        list(xTx = t(X) %*% X, xTy = t(X) %*% y)
    }

    chunks <- s3.ls(bucket)[, "bucket"]

    ans <- zmq.cluster.lapply(cluster = nodes, chunks, calc.xTx.xTy,
        y.transform = y.transform, x.transform = x.transform)
    exe.rpt <- attr(ans, "execution.report")
    xTx <- Reduce("+", lapply(ans, "[[", "xTx"))
    xTy <- Reduce("+", lapply(ans, "[[", "xTy"))
    ans <- solve(xTx) %*% xTy
    attr(ans, "execution.report") <- exe.rpt
    ans
}
```

distributed memory regression – more setup

What are we estimating?

We supply 'get.y' and 'get.x' which are applied to the each chunk.

'get.y' returns the log of the departure delay in minutes.
'get.x' returns a matrix of the departure time in hours and the log(distance) of the flight.

```
require(AWS.tools)
get.y <- function(X) {
    log(ifelse(!is.na(X[, "DepDelay"]) & X[, "DepDelay"] > 0, X[, "DepDelay"],
        1e-06))
}
get.x <- function(X) {
    as.matrix(cbind(round(X[, "DepTime"]/100, 0) + X[, "DepTime"]%%100/60,
        log(ifelse(X[, "Distance"] > 0, X[, "Distance"], 0.1))))
}
```

distributed memory regression – local execution

```
## research is an 8 core, 64GB server, 64/8 ~ 8GB per worker
## mongo is a 12 core, 32GB server, 32/12 ~ 2.6GB per worker
nodes <- c("research", "mongodb", "krypton")
coefs.local.runtime <- system.time(coefs.local <- distributed.reg(bucket =
"s3://airline.data",
    nodes, y.transform = get.y, x.transform = get.x))
print(as.vector(coefs.local))

## [1]  0.1702 -1.4925

print(coefs.local.runtime)

##    user  system elapsed
##    0.10    0.02  402.44

print(node.summary(attr(coefs.local, "execution.report")))

##        node jobs.completed
## 1  krypton              1
## 2  mongodb             12
## 3 research              8
```

distributed memory regression – cloud execution

Make sure you pick an instance type that provides adequate RAM per core.
m1.large is a 2 core machine with 7.5 GB of RAM 3.75GB per core (which is more than we need for this chunk size).

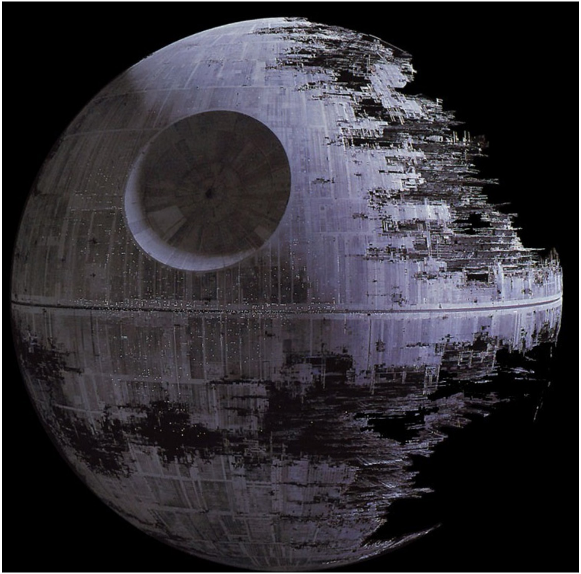
```
require(AWS.tools)
cluster <- startCluster(ami = "ami-1c05a075", key = "kls-ec2",
    instance.count = 11, instance.type = "m1.large")
nodes <- get.nodes(cluster)
coefs.ec2.runtime <- system.time(coefs.ec2 <- distributed.reg(bucket =
"s3://airline.data",
    nodes, y.transform = get.y, x.transform = get.x))

## turn the cluster off
res <- terminateCluster(cluster)
print(as.vector(coefs.ec2))

## [1]  0.1702 -1.4925

print(coefs.ec2.runtime)

##    user  system elapsed
##   0.152   0.088 277.166

print(node.summary(attr(coefs.ec2, "execution.report")))

##                  node jobs.completed
## 1   ip-10-114-150-31              2
## 2  ip-10-116-235-189              1
## 3   ip-10-12-119-154              2
## 4   ip-10-12-123-135              2
## 5  ip-10-204-111-140              2
## 6   ip-10-204-150-93              2
## 7    ip-10-36-33-101              2
## 8    ip-10-40-62-117              2
## 9   ip-10-83-131-140              2
## 10  ip-10-85-151-177              2
## 11   ip-10-99-21-228              2
```

Final points

- consuming cloud resources should not require arduous configuration steps (in contrast to MPI, SLURM, Sun Grid Engine, etc.)
- deathstar allows cloud access with minimal headache maximum simplicity

Conclusion

The next generation of R programmers needs something better than MPI... Please help me continue to build that tool.



Thanks!

Many people contributed ideas and helped debug work in progress as the package was being developed.

Kurt Hornik for putting up with my packaging.
John Laing for initial debugging.
Gyan Sinha for ideas and initial testing.
Prof Brian Ripley for just being himself.
My wife for enduring many solitary hours while I built these packages.