High Performance Computing with R

Pragnesh Patel and George Ostrouchov
pragnesh@utk.edu
Remote Data Analysis and Visualization(RDAV)
10 May 2011

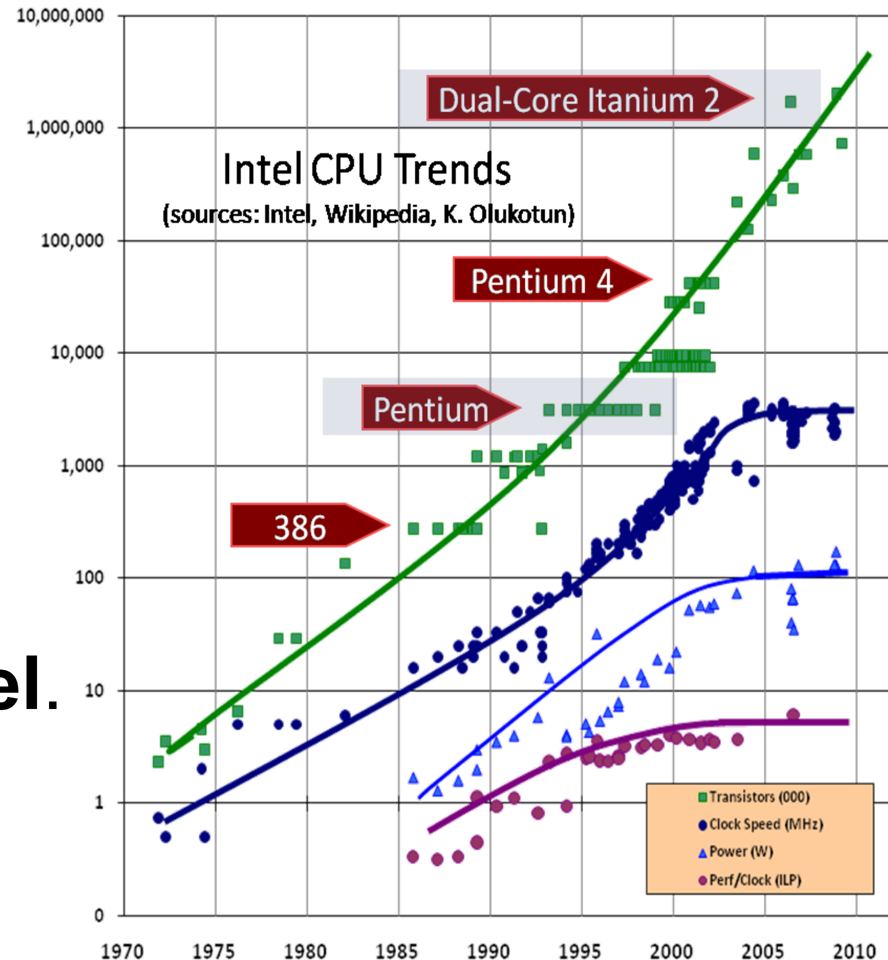
NIMBioS

Outline:

- Motivation
- Classes of parallel computers
- Multicore computing(MKL, pnmath, foreach, multicore, doMC)
- Cluster computing(Rmpi)
- GPU computing(gputools)
- R limitation and bigmemory
- mapReduce
- Rcpp and inline
- R profiling
- Case study in brief and lessons learned
- R-OpenMP project
- R-2.13 new features(OpenMP support and Byte code compilation)
- Other useful packages and links
- Summary
- References

Motivation:

- Clock speed saturates at 3 to 4 GHz.
- End of the free lunch.
- Computational intensive models in R.
- Large datasets.
- **So, the future is parallel.**



Introduction:

- Need to understand parallel programming paradigms in HPC.
- Need to understand computer architecture and its implication on parallel computing models.
- Choose the right tool for time consuming tasks depending on the type of application as well as the available hardware.

Classes of parallel computers:

- Multicore computing
- Cluster computing
- GPU computing

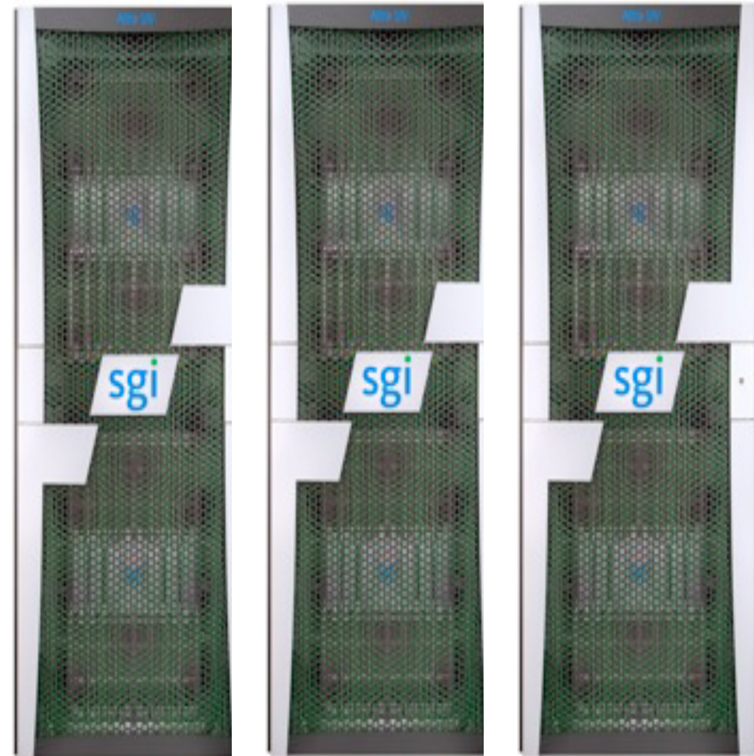
- Reconfigurable computing with FPGA
- Vector processors
- Distributed computing
- And many others…

How to get benefits of resources:

- R provides high level abstractions.
- R provides dynamics libraries and packages.
- R provides modularization.
- R provides mixing of programing paradigms.
- You can write multicore, cluster and GPGPU accelerated applications in R.

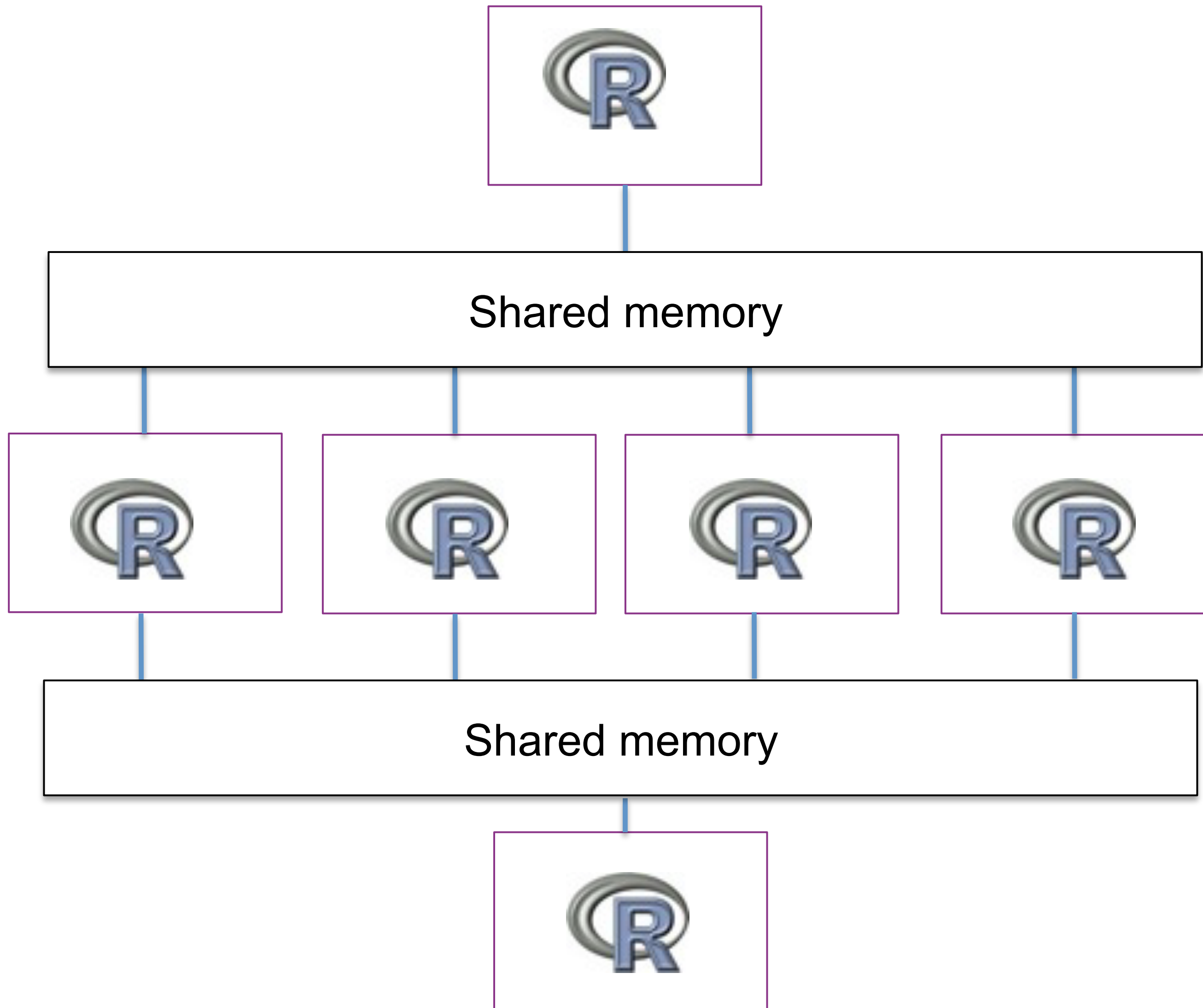
SMP:

- Multiple processors which share one global memory(RAM)
- Bus interconnect
- Threaded programs
- Communication via shared variables
- Easy to program
- SMPs are Commonplace because of multicore CPUs.
- Example: Nautilus



Reference: http://www.nics.tennessee.edu/computing-resources/nautilus

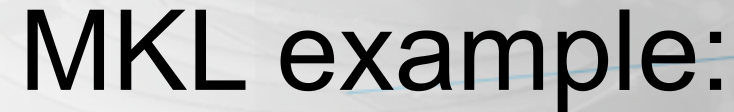
SMP and R:



Portion of R, lightweight compare to Rmpi processes

MKL:

- BLAS are standard building blocks for linear algebra. Highly-optimized libraries exist that can provide considerable performance gains.
- R can be built using so-called optimized Blas such as Atlas ('free'), Goto (not 'free'), or those from Intel or AMD; see the 'R Admin' manual for more information.
- Requires NO(very trivial) changes to serial code.
- Yet delivers good performance.



MKL example:

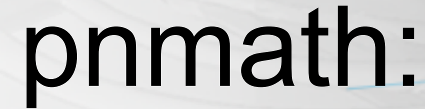
```
export MKL_NUM_THREADS=8
export MKL_DYNAMIC=FALSE

its = 2500
dim = 1750
X = matrix(rnorm(its*dim),its, dim)

system.time({C=matrix(0, dim, dim);for(i in 1:its)C = C + (X[i,]
%o% X[i,])}) # single thread breakup calculation

system.time({C1 = t(X) %*% X}) # single thread - BLAS matrix
mult

system.time({C2 = crossprod(X)})# single thread - BLAS matrix
mult

print(all.equal(C,C1,C2))
```

MKL results:

```
(1) user  system elapsed     # single thread breakup calculation
    74.540  7.628  83.274

 (2)  user  system elapsed   # single thread - BLAS matrix mult
using %*%
      2.316   0.092   2.410

 (3)  user  system elapsed   # single thread - BLAS matrix mult
using crossprod
      1.280   0.016   1.300

 (4) user  system elapsed    # multithreaded- BLAS matrix mult
with 8 threads using %*%
      2.188   0.020   0.367

 (5) user  system elapsed    # multithreaded- BLAS matrix mult
with 8 threads using   crossprod
      1.500   0.020   0.189
```

MKL benchmark results:

	1	2	4	8	16	32	64	128
Creation, transp.., deformation of a 2500*2500 matrix	**1.15**	**1.05**	**1.12**	**1.05**	**1.05**	**1.07**	**1.11**	**1.05**
2400*2400 normal distributed random matrix ^ 1000	**0.54**	**0.52**	**0.52**	**0.52**	**0.52**	**0.52**	**0.52**	**0.52**
Sorting of 7,000,000 random values	**1.37**	**1.37**	**1.37**	**1.37**	**1.36**	**1.37**	**1.37**	**1.37**
2800* 2800 cross-product matrix (b=a' * a)	**3.81**	**3.83**	**2.13**	**1.42**	**1.33**	**1.68**	**1.75**	**2.69**
Linea regression over a 3000*3000 matrix (c = a\b')	**1.61**	**1.88**	**0.89**	**0.61**	**0.49**	**0.53**	**0.87**	**1.30**
----------- Trimmed geom. Mean	**1.37**	**1.39**	**1.11**	**0.96**	**0.90**	**0.92**	**1.09**	**1.23**
FFT over 2,400,000 random values	**1.00**	**0.97**	**1.00**	**0.98**	**0.99**	**0.98**	**0.99**	**0.99**
Eigen values of a 640*640 random matrix	**0.89**	**1.81**	**0.96**	**0.91**	**1.01**	**0.98**	**1.17**	**1.30**
Determinant of a 2500*2500 random matrix	**1.51**	**1.78**	**0.95**	**0.59**	**0.55**	**0.35**	**0.42**	**0.30**
Cholesky decomposition of a 3000*3000 matrix	**1.42**	**1.64**	**0.75**	**0.52**	**0.42**	**0.38**	**0.46**	**0.58**
Inverse of a 1600*1600 random matrix	**1.29**	**1.65**	**0.90**	**0.64**	**0.29**	**0.62**	**0.71**	**3.80**
----------- Trimmed geom. Mean	**1.22**	**1.69**	**0.94**	**0.70**	**0.61**	**0.61**	**0.69**	**0.91**
3,500,000 Fibonacci numbers calculation (vector calc)	**1.05**	**1.02**	**1.03**	**1.03**	**1.28**	**1.03**	**1.26**	**1.40**
Creation of a 3000*3000 Hilbert matrix (matrix calc)	**0.76**	**0.74**	**0.74**	**0.78**	**1.21**	**0.78**	**1.21**	**1.47**
Grand common divisors of 400,000 pairs (recursion)	**2.82**	**2.77**	**2.79**	**2.79**	**5.17**	**2.79**	**5.18**	**6.85**
Creation of a 500*500 Toeplitz matrix (loops)	**1.08**	**1.06**	**1.08**	**1.09**	**1.26**	**1.07**	**1.26**	**1.39**
Escoufier's method on a 45*45 matrix (mixed)	**0.70**	**1.48**	**1.65**	**0.70**	**0.85**	**0.92**	**0.68**	**0.70**
----------- Trimmed geom. Mean	**0.95**	**1.17**	**1.22**	**0.96**	**1.25**	**1.00**	**1.24**	**1.42**



pnmath:

- It uses the OpenMP parallel processing directives for implicit parallelism.
- Loading the package replaces the built-in math functions by the parallel versions. At load time a calibration is carried out to determine the parallel overhead.
- It implements parallelized versions of most of the non-RNG routines in the math library.
- Requires NO(very trivial) changes to serial code.
- Can use OMP_NUM_THREADS environment variable to set number of threads.

pnmath example:

- Achieved speedup up to 650x.

```
>library(pnmath)
>t1<-system.time(sqrt(m))[3] # m is a vector
>t2<-system.time(exp(m))[3]
>t3<-system.time(qtukey(m,2,3))[3]
```

pnmath results(qtukey):

10 20 30 40 50 100 200 300 400 500 1000 2000 5000 10000 25000 50000 1e+05 5e+05 1e+06

stime

n

foreach and parallel backend:

- foreach: provides a method similar to for-loops for executing R expressions sequentially or in parallel.

```
>library(foreach)
>foreach(i=1:10) %dopar% sample(c("H", "T"),
10000,replace=TRUE)
```

Warning message:

executing %dopar% sequentially: no parallel backend registered

- Must register a parallel backend to manage the parallel execution of the loop.
- Backend: doMC, doMPI, doSNOW, doSMP

doMC and multicore:

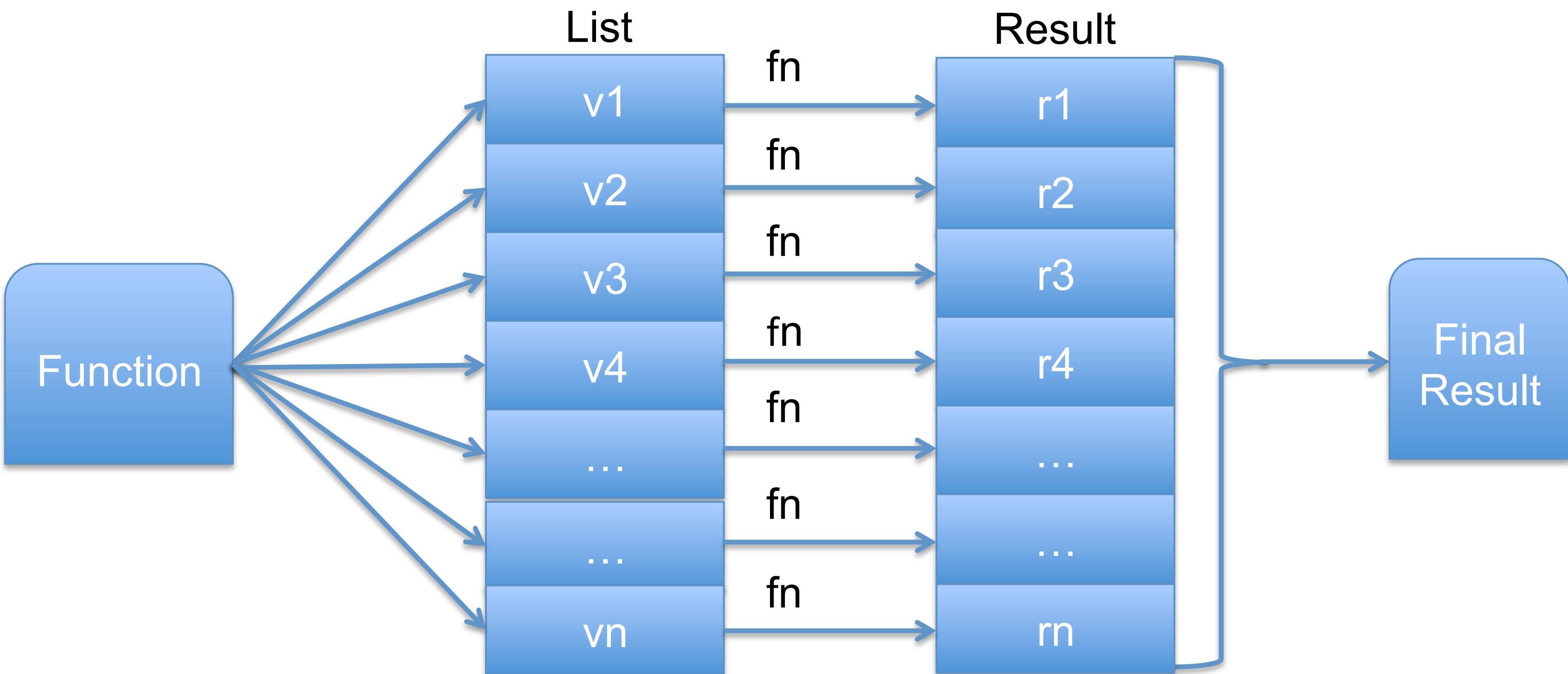
- doMC: parallel multicore back end for use with the foreach package.
- multicore: provides a way of running parallel computation in R on machines with multiple cores or CPUs.
  - `mclapply`: parallelized version of `lapply`.
  - `parallel`: evaluates an expression asynchronously in a separate process.
  - `pvec`: parallelizes the execution of a function on vector elements by splitting the vector and submitting each part to one core.

foreach and doMC example:

```
# R
> library(foreach)
> library(doMC)
> registerDoMC(cores=4)

> system.time(foreach(i=1:10) %do% sum(runif(10000000)))

   user  system elapsed
   4.796   0.448   5.245

> system.time(foreach(i=1:10) %dopar% sum(runif
(10000000)))

   user  system elapsed
   4.332   0.609   1.459
```

R lapply:

- Natural candidate for automatic parallelization.



- Examples: `multicore(mclapply),`
  `Rmpi(plapply)`

mclapply example:

```
# R
>library(multicore)
>multicore::detectCores()
>options(cores = 8)
>getOption('cores’)
>test <- lapply(1:10,function(x) rnorm(10000))

>system.time(x <- lapply(test,function(x) loess.smooth
(x,x)))

   user  system elapsed
  0.664   0.176   1.407

>system.time(x <- mclapply(test,function(x) loess.smooth
(x,x)))

    user  system elapsed
  0.008   0.008   0.351
```

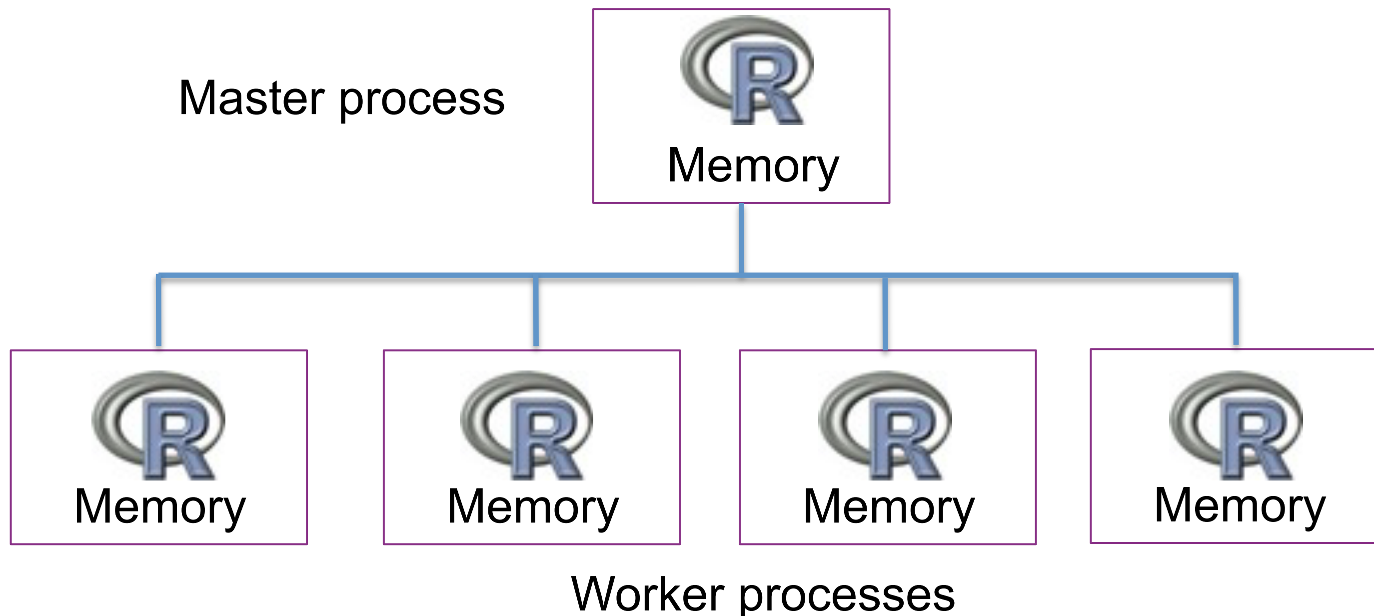
Cluster computing:

- Distributed memory
- Ethernet connect, Infiband connect
- Better scalability
- Message passing interface
- Example: Kraken

Reference: http://www.nics.tennessee.edu/computing-resources/kraken

Rmpi:

- Rmpi provides interface to MPI APIs.
- R is required at each compute node.
- Supports many MPI standard functions.
- Require Parallel programming knowledge.



Rmpi example:

```
# Load the R MPI package if it is not already loaded.
if (!is.loaded("mpi_initialize"))
    library("Rmpi”)
}
# Spawn as many slaves as possible
mpi.spawn.Rslaves()

# In case R exits unexpectedly, have it automatically clean up
# resources taken up by Rmpi (slaves, memory, etc...)
.Last <- function(){
 if (is.loaded("mpi_initialize")){
   if (mpi.comm.size(1) > 0){
        print("Please use mpi.close.Rslaves() to close slaves.”)
        mpi.close.Rslaves()
   }
   print("Please use mpi.quit() to quit R")
  .Call("mpi_finalize”)
 }
}
```

Rmpi example continue:

```
# Tell all slaves to return a message identifying
themselves
 mpi.remote.exec(paste("I am",mpi.comm.rank
(),"of",mpi.comm.size()))

#mpi.remote.exec() actually is sending a message to every
slave asking it to execute the given code, and each child
is sending a message back to the master with the result.

 # Tell all slaves to close down, and exit the program
 mpi.close.Rslaves()
 mpi.quit()
```

Rmpi example output:

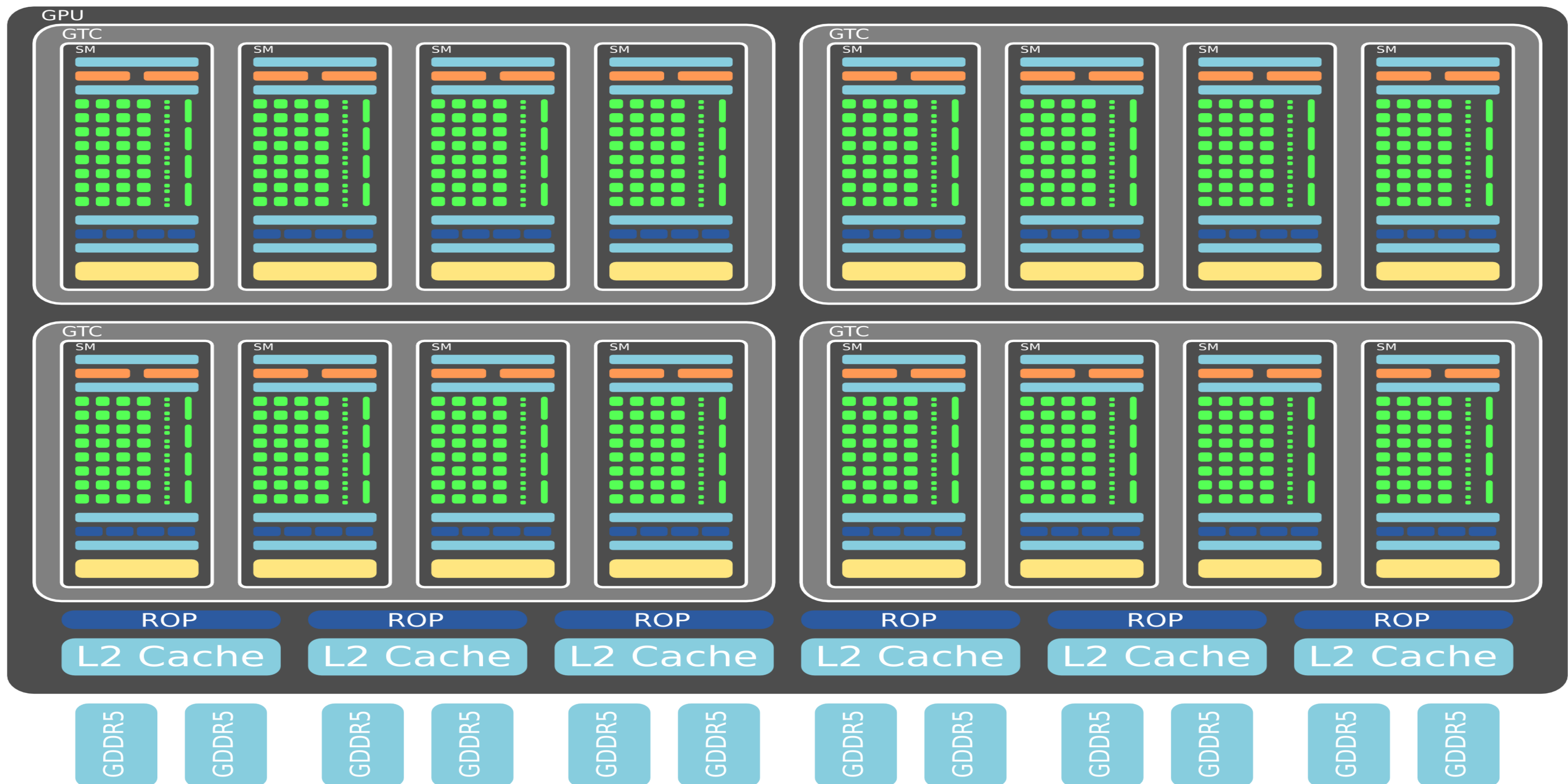
```
>mpi.spawn.Rslaves()
master (rank 0, comm 1) of size 8 is running on: nautilus
slave1 (rank 1, comm 1) of size 8 is running on: nautilus
slave2 (rank 2, comm 1) of size 8 is running on: nautilus
slave3 (rank 3, comm 1) of size 8 is running on: nautilus
…….
…….
># Tell all slaves to print out a message identifying themselves
>mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
$slave1
[1] "I am 1 of 8”
$slave2
[1] "I am 2 of 8”
$slave3
[1] "I am 3 of 8”
$slave4
[1] "I am 4 of 8”
……
```

GPU computing:

- Special-purpose coprocessor for graphics application.
- GPU architecture are specialized for computer intensive, highly-parallel computation, and therefore are designed such that more resources are devoted to data processing than caching and flow control.
- Shared memory, typically 100s of core.
- CUDA and OpenCL programming models.

GPU architecture :

- High level block diagram of NVIDIA GPU chip.



Reference: http://nvidia.com

GPU memory model:

- Multilevel levels of memory hierarchy

GPU

GTC GTC GTC GTC

SM

ROP ROP ROP ROP ROP ROP

L2 Cache L2 Cache L2 Cache L2 Cache L2 Cache L2 Cache

GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5 GDDR5

L2 Cache

Global Mem / Local Mem

SM

I-Cache

Warp Sched Warp Sched

Constant Cache

Register File

PE PE PE PE LD/ST LD/ST SFU

Xbar

Shared Mem / L1

tex tex tex tex

texture cache

PolyMorph Engine

Constant Cache

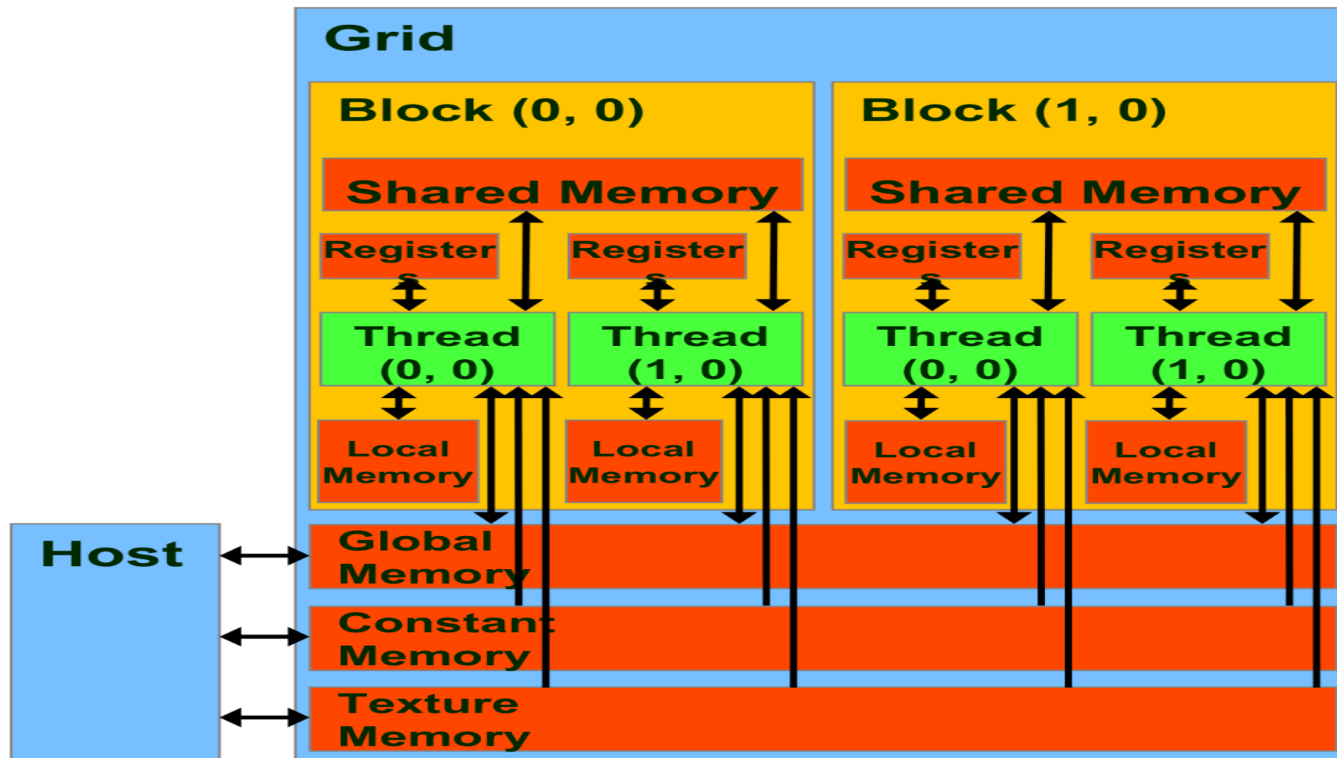
Shared Mem / L1 Cache

Texture Cache

Reference: http://nvidia.com

GPU memory model:

- GPU has much more aggressive memory subsystem.

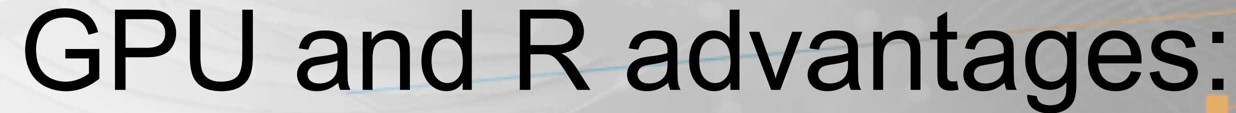


Reference: http://nvidia.com

gputools:

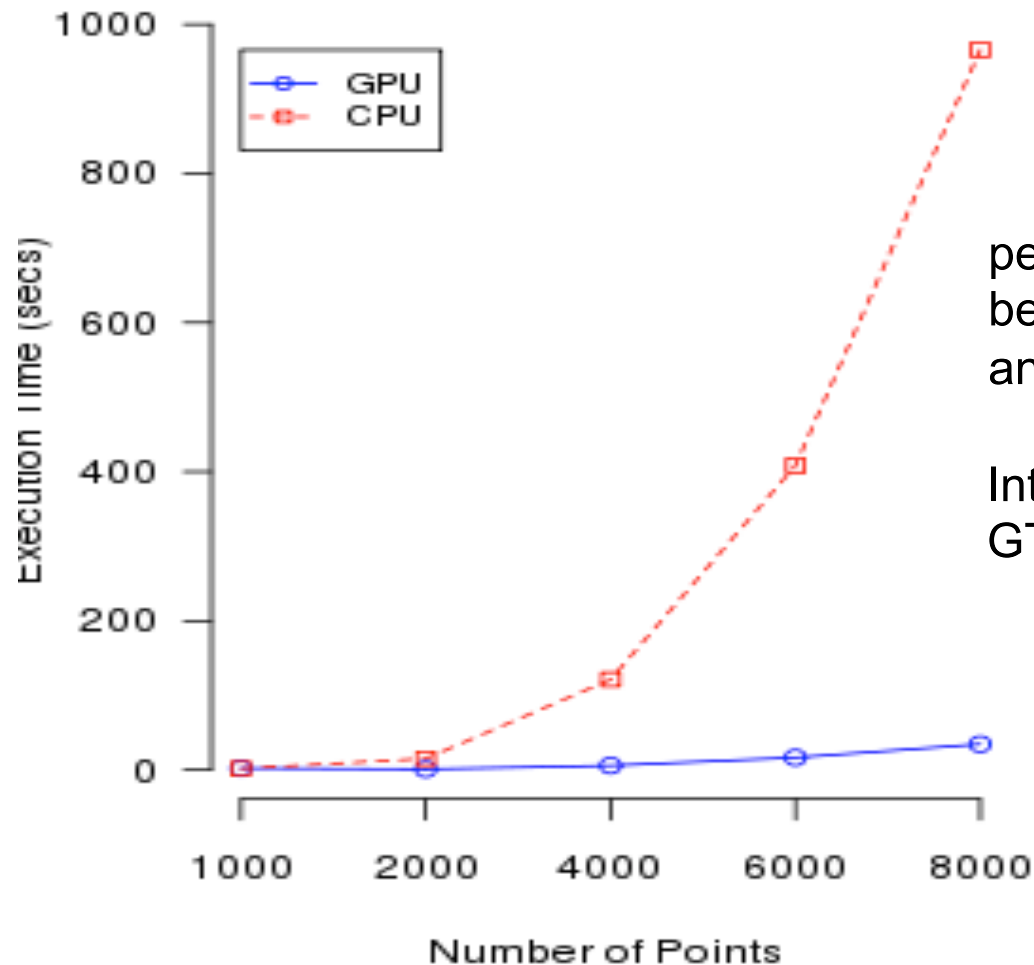
- It provides R interfaces to handful common statistical algorithms.
- Implemented using mixture of CUDA language, CUBLAS library and CULA library.
- It contains many other functions: Hierarchical clustering, SVM training, SVD, Least-squares fit, linear modeling etc…
- Less-communicative algorithms seeing speedups over 20x on data set of moderate size (e.g. Hierarchical cluster >20x).
- Speedup factors vary with CPU, memory configurations and, of course, GPU.

GPU and R advantages:



```
>library(gputools)
>matA <- matrix(runif(3*2), 3, 2)
>matB <- matrix(runif(3*4), 3, 4)
>gpuCrossprod(matA, matB) # Perform Matrix Cross-product
with a GPU


>numVectors <- 5
>dimension <- 10
>Vectors <- matrix(runif(numVectors*dimension),
>numVectors, dimension)
>gpuDist(Vectors, "euclidean")
>gpuDist(Vectors, "maximum")
>gpuDist(Vectors, "manhattan")
>gpuDist(Vectors, "minkowski", 4)
```

gputools benchmark:



performance comparison between the function 'hclust' and ‘gpuHclust’ function.

Intel Core i7 single thread vs GTX 260(192 cores)

Reference: http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/

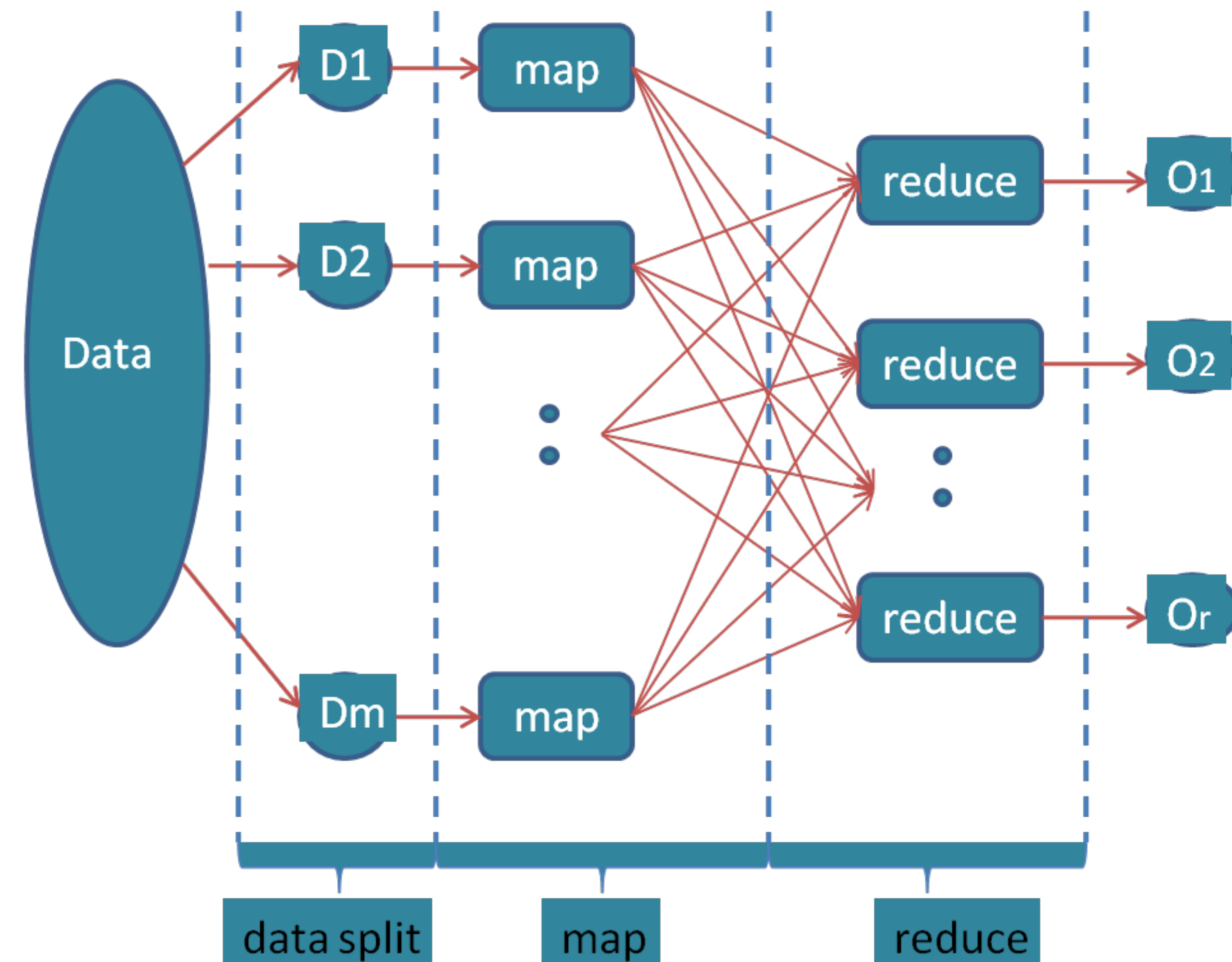
R limitation and bigmemory:

- R is a memory bound language.
  - 32 bit integer indexing limit.
- Multi-gigabyte data sets often frustrate R users.
- bigmemory, biganalytics, bigalgebra, bigtabulate implement massive matrices and support manipulation and exploration.
- The data structures may be allocated to shared memory, allowing separate processes on the same computer share access to single copy of the date set.
- The data structures may also be file-backend allowing users to easily manage and analyze data sets larger than available RAM and share them across nodes of a cluster.

bigmemory and other packages:

- bigmemory: supports the creation, manipulation and storage of large matrices.
- bigalgebra: provides linear algebra functionality with large matrices.
- biganalytics: extends the functionality of bigmemory.
- bigtabulate: supports table(), split() and tapply() like functionality for large matrices.
- foreach + bigmemory: a winning combination for massive data concurrent programming.

Map Reduce:



- The framework supports the splitting of data.
- Outputs of the map functions are passed to the reduce functions.
- The framework sorts the inputs to a particular reduce function based on the intermediate keys before passing them to the reduce function.
- An additional step may be necessary to combine all the results of the reduce functions.

Map Reduce:

- MAP step: The master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

- REDUCE step: The master node then takes the answers to all the sub-problems and combines them in some way to get the output - the answer to the problem it was originally trying to solve.

mapReduce:

- mapReduce is an algorithm provides a simple framework for parallel computations. This implementation provides (a) a pure R implementation (b) a syntax following the mapReduce paper and (c) flexible and parallelizable back end.
- MapReduce is a framework for processing huge datasets on a large number of computers (cluster, grid or cloud)
- Nothing more than `apply(map(data), reduce)`

Recap parallel R packages:

- MKL/pnmath
- foreach, doMC
- multicore

- Rmpi

- gputools

- bigmemory
- mapReduce

Rcpp and inline:

- Rcpp: facilitates the integration of R and C++.
  - All R types are supported.
  - The mapping of data types works in both directions.
- inline: provides functionality to dynamically define R functions and S4 methods with in-lined C, C++ and Fortran.
  - `cfunction`: inline C, C++, Fortran function calls from R.
  - Help to improve the performance of computational intensive functions.

R profiling:

- Profiling a program means determining how much execution time a program spends in various different sections of code.
- We need to know where our code spends the time to takes to compute our tasks.
- R provides the tools for performance analysis.
  - The `system.time` function.
  - The `Rprof` for profiling R code.
  - The `Rprofmem` function for profiling memory usage.
- In addition, the `profr` and `proftools` package on CRAN can be used to visualize `Rprof` data.

R profiling:

```
Rprof("boot.out")
##your code
Rprof(NULL)

##generates boot.out file

Then run > R CMD Rprof boot.out
```

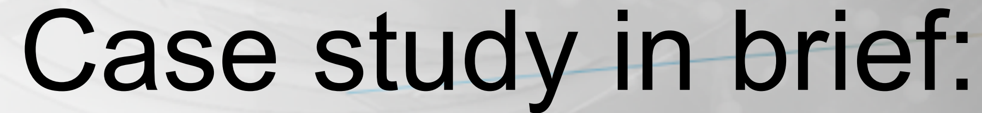
- It does impose small performance penalty.

R memory profiling:

- R has to compile with "`--enable-memory-profiling`" option.
- Difficult to use because of R garbage collector. Memory is allocated at well-defined times in an R program but is freed whenever the garbage collectors happens to run.

```
Rprofmem("boot.out")
##your code
Rprofmem(NULL)
##Generates boot.out file
```

Case study in brief:



- Working on Prof. Michael's code.
- To find the MLEs for all the amino acids under a given value.
- It parallelized across genes for each amino acid.
- It uses mclapply function from multicore package.
- One round robin iteration takes about 1 days.

Nested for loops:

```
#Calculate the MLE of parameters under the hypergeomtric
approximation
calc_hypergeo_mle_mult_indx <- function(mult_indx_signs)
{
   #Starting points of parameters delta_t under hypergeometric
approx.
                                                       160 iteration
   for()
   {
        for()
        {
          for()
          {
           optimum <- newuoa(initial_par, wrap_hypergeo, aa=i,
cod_pairs=cod_pairs, control=list(maxfun=maxiter));
          }
            ## single call of newuoa calls wrap_hypergeo function
50-70 times.
        }
   }
}
```

Total number of calls (wrap_hypergeo)= 160 * 50-70 = ~ 8000-10000

Parallel wrapper function:

```
# Parallelization wrapper for hypergeometric approximation
wrap_hypergeo <- function(par, aa, cod_pairs, hess=FALSE)
{

  stime <- system.time(tmpout <- mclapply(gindx, function(x)
        {hypergeo_llk(i=x, time=tmpetime, mut=tmpmut, aa=aa,
        cod_pairs=cod_pairs)}, mc.cores=Ncores, mc.presche\
        dule=TRUE))

  ## iterates over dataset
}
```

2500 sequences

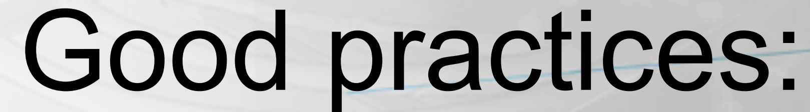
	22 cores	128 cores	256 cores
Calc_hypergeo_mle_mult_indx	**72 minutes**	**380 minutes**	**> 11 hours**
Wrap_hypergeo (single instance exec)	**1 second**	**4.8 seconds**	**8.9 seconds**

Good practices:

- Loop fission: technique attempting to break a loop into multiple loops over the same index range but each taking only a part of the loop's body.
- Often it may be the case that you have a main loop in your code, perhaps updating many matrices.
- But, it could be that there is no interdependence amongst the matrices you are updating.

```
for(variable in sequence){
 m1[ ]=
 m2[ ]=
}
```

Good practices:



- Break down large loop body into smaller ones to achieve better utilization of locality of reference.
- This second approach can often yield a reasonable gain in a very long, intensive loop.
- Real compilers (i.e. C, Fortran, ...) do this automatically, but R does not.

```
for(variable in sequence){
  m1[ ]=
}
For(variable in sequence){
  m2[ ]=
}
```

Good practices: vectorization

- Vectorization makes loops implicit in expression.
- Replacing the loop yielded a gain of a factor of more than 35.

```
> sillysum <- function(N) { s <- 0;
+ for (i in 1:N) s <- s + i; return(s)
}
> system.time(print(sillysum(1e7)))
[1] 5e+13
  user  system elapsed
  7.288   0.504   7.873

> system.time(print(sum(as.numeric(seq(1,1e7)))))
[1] 5e+13
  user  system elapsed
  0.096   0.124   0.218
```

R-OpenMP project:

- OpenMP
  - It is a shared memory model.
  - It is a Lightweight approach.
  - Workload is distributed between threads.
  - Supported by many compilers: GNU, Intel, IBM, NAG and PGI.
- Translation of R functions to C/Fortran functions.
- It will provide easy programmability to users to use multicore architecture.

R-OpenMP detail:

```
>registerDoFortan (”ifort -openmp -g -O3”)
>myfunc<- foreach (i=1:n, x=double(n), y=double
(n), .combine=”+”) %dopar% {y[i]<-sin(x[i])+3*cos(2*x[i])}
```

Then generates a fortran file containing a fortran version of the subroutine:

```
subroutine myfunc (integer n, double x, double y)
double x (n), y (n)
!$OMP DO
do i=1,n
y (i)=sin (x(i))+3*cos(2*x(i))
enddo
end subroutine
```

Then the fortran code is compiled on the fly and imported as a shared

object into R:

```
> dyn.load(“myfunc.so”)
```

R 2.13 new features:

- Support for packages which wish to use OpenMP.
- Byte compiler: Compiles R code to a `byte code' representation.
  - To compile all the base and recommended packages, run `make bytecode`.

Useful links and Packages:

- magma: Matrix Algebra on GPU and Multicore Architectures.
- snow: Simple networks of workstations.
- http://cran.r-project.org/web/views/HighPerformanceComputing.html by Dirk Eddelbuettel
- http://www.revolutionanalytics.com/subscriptions/docs/RevolutionREnterprise4.0/parRman.pdf

Summary/Wrapping up:

- In this tutorial session, we covered
  - Classes of parallel computers
  - MKL, pnmath, foreach, multicore, doMC
  - Rmpi
  - gputools,
  - bigmemory, mapReduce
  - Profiling
  - Rcpp and inline
  - Case study
  - Good practices
  - R-OpenMP project
  - R-2.13 new features

References:

- http://dirk.eddelbuettel.com/papers/useR2009hpcTutorial.pdf
- http://www.lrz.de/services/compute/courses
- http://cscads.rice.edu/workshops/summer09/slides/analysis-visualization/nagiza-samatova-cscads-2009.pdf
- http://cran.r-project.org/web/views/HighPerformanceComputing.html by Dirk Eddelbuettel
- Implicit and Explicit Parallel Computing in R by Luke Tierney
- http://www.compbiome.com/2010/04/r-parallel-processing-using-multicore.html
- http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/
- http://labs.google.com/papers/mapreduce.html
- http://math.acadiau.ca/ACMMaC/Rmpi

Thank You !!!