How to speed up your R computation by vectorization and parallel programming

Lecture 4

1. Introduce MPI
2. How to use Rmpi's parallel apply functions to do "embarrassingly parallel"
3. Introduce some advanced MPI programming

Introduce MPI

- Software for parallel computing
  - ⋆ Low level programming: C, C++, Fortran
  - ⋆ MPI: Message-Passing Interface
  - ⋆ PVM: Parallel Virtual Machine
  - ⋆ Both were developed in early 1990's
- What is the Message-Passing?
  - ⋆ On a Beowulf cluster, a process can only access the local memory it sits in

- Message-Passing: a set of processes running simultaneously that have only local memory but are able to communicate with other processes by sending and receiving messages
- One of message-passing features is that data transfer from the local memory of one process to the local memory of another requires operations to be performed by both processes

- Advantages of the Message-Passing model
  - Universality: fits well on supercomputers as well as Beowulf clusters
  - Expressivity: express parallel algorithms well (compare to true parallel programming)
  - Ease of debugging: compare to true parallel programming
  - Performance: each process can take advantages of local machine hardware
- MPI is just a library to C, C++, and Fortran
  - MPI is not a language
  - doesn't change any C, C++, and Fortran programming (syntax, compiling, etc)
  - MPI functions can added to existing C, C++, or Fortran codes to make them use parallel environments such as Beowulf clusters
  - MPI-1 standard has 129 functions
  - Combined MPI-1 and MPI-2 standard has 287 functions

- There are several implementations of MPI: MPICH, MPICH2, LAM-MPI, OpenMPI, CARYMPI, INTELMPI, etc
- All merge to OpenMPI. On Mac OS X one needs to install OpenMPI
- On Windows platform, Microsoft MPI is available
- Rmpi is an R interface to MPI
  - MPI C-level functions are ported to R functions
  - Rmpi loads MPI environment automatically
  - Each running R (master or slave) is a stand alone process with MPI capabilities: inter R communication capabilities
  - Rmpi is capable of exchange complicated datasets
  - Rmpi still allows lower level programming or interacts with other non R processes
- MPI initialization and finalization
  - When Rmpi is loaded, R changes from a regular process to be a MPI aware process
  - Each MPI program must run finalization to release resource back to system
    To quit R, use mpi.quit()
- MPI communicators
  - MPI is primarily a communication library

- ⋆ Once a MPI program initialized, it must be associated with a communicator
  - ⋆ What is a communicator?
  - ⋆ A communicator represents a group in which MPI programs (processes) can make communication with each other
  - ⋆ Every process within a communicator has unique rank in the range from 0, 1, ..., size-1 (size=total number of processes within a communicator)
  - ⋆ In Rmpi, a communicator is represented by an integer 0, 1, 2, ...
  - ⋆ There are many Rmpi functions replying on a communicator to make a communication
  - ⋆ The default communicator in Rmpi is 1
  - ⋆ R-functions: mpi.comm.size, mpi.comm.rank
- On high level there are two types of communications: blocking and nonblocking communcations
  - ⋆ A blocking MPI call means that the program execution will be suspended until the message is sent or received
- Within a communicator, there are two kinds of communications:
  - ⋆ point-to-point call and collective call
- MPI point-to-point communications

- ⋆ MPI send and receive procedures are the building blocks of MPI programs
- ⋆ R-functions: mpi.send and mpi.recv
- ⋆ It involves the transmission of a message from a sender's memory to a receiver's memory
- ⋆ The location, size, data type must be specified
- ⋆ In R, size is not needed since length(obj) is used
  Data type: 1=double, 2=integer, 3=char (string)
- ⋆ A tag is used to specify a specific message
- ⋆ Wildcard values may be used to receive messages from any source and/or any tag: mpi.any.source() and mpi.any.tag()
- ⋆ It is important that the receiver's memory has enough room to receive a message
- ⋆ How does a receiver prepare its mailbox for the incoming message if it does not know the size of an incoming message?
- ⋆ R-function: mpi.probe
- ⋆ mpi.recv and mpi.probe use MPI_Status structure: information about the actual message received
- ⋆ In R, MPI_Status is represented by a nonnegative integer (default=0)
- ⋆ mpi.probe(source, tag, comm = 1, status = 0): given source and tag, the incoming

message information is save into status=0

  - ⋆ mpi.get.sourcetag(status=0) returns the source and tag
  - ⋆ mpi.get.count(status=0) returns the exact length of an incoming message
  - ⋆ mpi.recv has the status argument so that mpi.get.sourcetag and mpi.get.count can be used
  - ⋆ How is a complicated data object (e.g. list) sent out or received?
  - ⋆ Answer: any R object can be (un)serialized as a raw vector (up to 4GB)
  - ⋆ Examine mpi.send.Robj and mpi.recv.Robj

- MPI collective communications
  - ⋆ Question: A member wants to send a dataset to all members within a communicator?
  - ⋆ One solution: The sending member

    ```
    for (in 1:size)
        mpi.send.Robj(dataset, dest=i,...)
    ```

    All receiving members

    ```
    mpi.recv.Robj(source=0,...)
    ```

  - ⋆ Use collective communications

    all members within a communicator must do the same
  - ⋆ R-fuction: mpi.bcast

- ⋆ mpi.bcast is a blocking call: all members within a communicator must execute it together otherwise there is a deadlock
- ⋆ Other MPI collective functions:
  mpi.abort
  mpi.allgather
  mpi.allgatherv
  mpi.allgather.Robj
  mpi.barrier
  mpi.gather
  mpi.gatherv
  mpi.gather.Robj
  mpi.reduce
  mpi.scatter
  mpi.scatterv
  mpi.scatter.Robj
- ⋆ Extensions to R:
  mpi.bcast.Robj2slave

mpi.bcast.data2slave
mpi.bcast.Rfun2slave
mpi.bcast.cmd
mpi.close.Rslaves (match with mpi.spawn.Rslaves)
mpi.remote.exec

How to use Rmpi's parallel apply functions to do "embarrassingly parallel"

- Spawning R slaves
  - ⋆ When Rmpi is loaded, there is only one R (master)
  - ⋆ Need to spawn R slaves to do the real computations
  - ⋆ R function: mpi.spawn.Rslaves(). It will spawn mpi.universe.size() (=n) number of R slaves
  - ⋆ In total, there are one master (rank 0) and n slaves (ranks 1 to n) associated with communicator 1
  - ⋆ All slaves are standalone R processes and are "naked"
  - ⋆ Note: on a big cluster like SHARCNET, spawning is prohibited. Different way is used to create slaves

- Parallel Random Number Generator
  - Need RNG so that each node or cpu has independent stream of random numbers
  - Need large cycle so that it will not repeat quickly
  - Just need a uniform parallel random number generator
  - Other distributions can be implemented by uniform RNG
  - R function RNGkind
  - Rmpi uses R's rlecuyer RNG for parallel RNG
  - R function: mpi.setup.rngstream
  - Implementations for given a random number generator
    - One seed for all members within a communicator

      This seed has a slightly different meaning than is conventionally used. It is an encoding of the starting state
    - Each member has a unique random number stream
    - Random numbers generated from different streams must be independent
    - Random numbers can be reproduced for given a seed and streams
- Simplest way to run a job on all slaves: mpi.remote.exec(fun())
  - mpi.bcast.Robj2slave may be used to move necessary data or functions to slaves
  - mpi.bcast.data2slave may be used to move a vector or matrix data without

(de)encoding process
  - ⋆ mpi.bcast.Rfun2slave may be used to move R functions to slaves
  - ⋆ mpi.comm.rank() can be used to identify which slave this job is running on
  - ⋆ Return values are either a vector or a matrix or a list
  - ⋆ Limitation: scalability
- Another way is to use mpi.apply
  - ⋆ mpi.apply is different from apply (an array instead of a matrix)
  - ⋆ See the source codes
  - ⋆ mpi.bcast.cmd is particularly useful to let all slaves execute a command without deadlock
  - ⋆ try function is used to catch any errors
  - ⋆ do.call is used to execute a command represented by a string
- How to handle length(x) $>$ mpi.comm.size()-1?
  - ⋆ Load balancing approach
  - ⋆ Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time
    - $*$ CPUs could have different speed

    - Each job could take different time to finish
  - Check mpi.applyLB

- mpi.apply and mpi.applyLB are two building blocks to implement the following parallel apply functions
  - mpi.parApply – (Load balancing) parallel apply
  - mpi.parCapply – (Load balancing) parallel column apply
  - mpi.parLapply – (Load balancing) parallel lapply
  - mpi.parRapply – (Load balancing) parallel row apply
  - mpi.parReplicate – A wrapper to mpi.parSapply for repeated eval of an expr
  - mpi.parSapply – (Load balancing) parallel sapply
  - Those functions should be able to carry out any types of Monte Carlo simulation
  - Basic idea: .splitIndcies is used to partition entire array, matrix, or list into job.num small pieces
  - If job.num $>$ mpi.comm.size()-1, load balancing is used
  - For optimal load balancing, choose
    job.num=k*(mpi.comm.size()-1) with k=2,3, or 4
    Big k may lead to network overhead

Introduce some advanced MPI programming

- Nonblocking MPI
  - An MPI nonblocking call returns immediately after the call is initiated and does not wait to be certain that the communication buffer is safe to use. You must make sure that the send buffer has been copied out before reusing it
  - Nonblocking MPI functions in Rmpi
    - mpi.isend, mpi.irecv, mpi.isend.Robj, mpi.iprobe
    - mpi.cancel,mpi.test.cancelled,mpi.test, mpi.testall, mpi.testany, mpi.testsome, mpi.wait, mpi.waitall, mpi.waitany, mpi.waitsome
  - Nonblocking (on master only) parallel apply functions
    - mpi.iparApply, mpi.iparCapply, mpi.iparLapply, mpi.iparRapply, mpi.iparReplicate, mpi.iparSapply
- Task pull and push approach
  - Task push: Master pushes all tasks to slaves and wait results back in sequential order (may not work)
  - Task pull: Master pushes first round of tasks to each slave and collects a result and send a new task in FCFS order

- ⋆ All parallel functions in Rmpi use pull approach (in fact it is more or less like push and pull approach)
- ⋆ If slaves know the task list in advance, they can start the first round of task on their own
- ⋆ To save waiting on slave side, mpi.isend.Robj may be used

- Cartesian topology functions in Rmpi
  - ⋆ mpi.cart.create—create Cartesian structures of arbitrary dimension,
  - ⋆ mpi.cart.get—return dims, periods, and coords of calling process,
  - ⋆ mpi.cart.coords—return the coord for a given rank,
  - ⋆ mpi.cart.rank—return the rank for a given coord,
  - ⋆ mpi.cart.shift—find neighbor ranks.
- Parallel Computations of Response Surface Regression in R
  - ⋆ Let $T(\underline{X})$ be a statistic of estimating a unknown parameter θ, with sample $\underline{X} = (X_1, \ldots, X_n)$. Except some simple cases, we don't have the closed distribution form of $T(\underline{X})$. Often we reply on its asymptotic result for proper inference

$$a_n(T(\underline{X}) - \theta) \xrightarrow{\mathcal{D}} W,$$

where $a_n > 0$ and $a_n \to 0$.

- ⋆ Under a null hypothesis of interest, we use W to find critical values and/or p-values. This leads to inaccurate assessment of the hypothesis test when sample sizes are small.
- ⋆ MacKinnon (2002) proposed to use RSR simulation technique to find quantiles of $T(\underline{X})$ under the null hypothesis.
- ⋆ Simulation steps in RSR. We assume that $\underline{X}$ can be simulated under a null hypothesis repeatedly.
  - ∗ Choose a proper set of sample sizes, say, $n = 10, 20, \ldots, 90, 100, 200, \ldots, 900, 1000,$.
  - ∗ Choose a proper set of probabilities, say, $probs = 0.90, 0.95, 0.99$.
  - ∗ For each sample size n, compute N replications of $T(\underline{X})$ and find its corresponding quantiles at $probs$. Repeat the same procedure M times in order to find local variations of those quantiles.
- ⋆ Estimating RSR steps.
  - ∗ For each $probs$ and sample size n, compute sample mean and variance of quantiles, denote them as $\bar{q}(n, probs)$ and $v\bar{a}r(n, probs)$.
  - ∗ Set up the regression line

$$\bar{q}(n, probs) = k_0 + k_1 n^{-1/2} + k_2 n^{-1} + k_3 n^{-3/2} + k_4 n^{-2} + \text{error}.$$

  - ∗ Use R's lm to find RSR with $weights = 1/v\bar{a}r(n, probs)$ (weighted least squares).
- ⋆ In order to estimate quantiles reliably, N should be at least 10,000 (prefer 100,000). For the same reason, M should be at least 100 (prefer 200). With $N = 100,000$ and $M = 200$, total simulations at each simple size is $N * M = 20,000,000$.
- ⋆ Regression terms in RSR may differ for different types of statistics. For example, for unit root test, regression terms n^{-1}, n^{-2}, n^{-3} are preferred.
- ⋆ Significance checking of regression terms can be carried out in lm and should be uniformly selected for all different quantiles.
- ⋆ Implementation of RSR involves many N replications (jobs) which can be run independently on any of available R slaves. This is a typical "embarrassing parallel" job which is common for many Monte Carlo simulations.
- ⋆ R master and slaves must coordinate with each other to avoid any deadlocks or race conditions.
- ⋆ From R master point of view, jobs can be carried out through "push" them to slaves or "pull" them from salves. "push" is used to implement RSR.
- ⋆ When R master pushes jobs to slaves, load balancing must be considered. To achieve load balancing for RSR, master will push jobs with largest sample sizes first

to slaves.

- ⋆ Functions used in implementing RSR.
  - ∗ mpi.rsr is the main function used by end users.
  - ∗ slaves.rsr is used by all slaves.
  - ∗ mpi.bcast.Robj2slave is used to send all necessary R objects to slaves.
- ⋆ Flow charts of the function mpi.rsr
  - ∗ Tell slaves to run the function slave.rsr.
  - ∗ Send all necessary R objects to slaves.
  - ∗ Send first round of jobs with the largest sample sizes to slaves.
  - ∗ Have a while loop sending jobs to slaves. During a loop, collect a result and send out a new job with first come first reply policy (load balancing).
  - ∗ Collect final round of results and send signals to terminate each slave.
  - ∗ Return computed results.