# Rhpc: An R package for High-Performance Computing

Eiji NAKAMA [†] Junji NAKANO[‡]

[†] COM-ONE Ltd., Japan

[‡] The Institute of Statistical Mathematics, Japan

Workshop on Distributed Computing in R
Jan 26-27, 2015
HP Labs, Palo Alto, CA, USA

# Outline

# Outline

# Introduction

- The Institute of Statistical Mathematics possesses three supercomputers. Their types are all different: a shared memory system, a distributed memory system and a cloud system.
- We hope to use R on them efficiently. For this purpose, we have developed a package `Rhpc`, which stands for "R for High Performance Computing".
- `Rhpc` follows the way of `snow` package, and is tuned for just using MPI.
- We also noticed that NUMA (Non-Uniform Memory Access) support is important for present parallel computing on R.

# Outline

# Rhpc functions (1)

- Rhpc supports long vector, so it can handle "big data".
- Worker process is written by Embedding R.
- Functions to initialize and finalize MPI, and get MPI communication handle
  - Rhpc_initialize
  - Rhpc_getHandle (∼ snow::makeMPICluster)
  - Rhpc_finalize (∼ snow::stopCluster)
- Functions to call Rhpc_worker
  - Rhpc_worker_call (∼ snow::clusterCall)
  - Rhpc_Export (∼ snow::clusterExport)
  - Rhpc_EvalQ (∼ snow::clusterEvalQ)

# Rhpc functions (2)

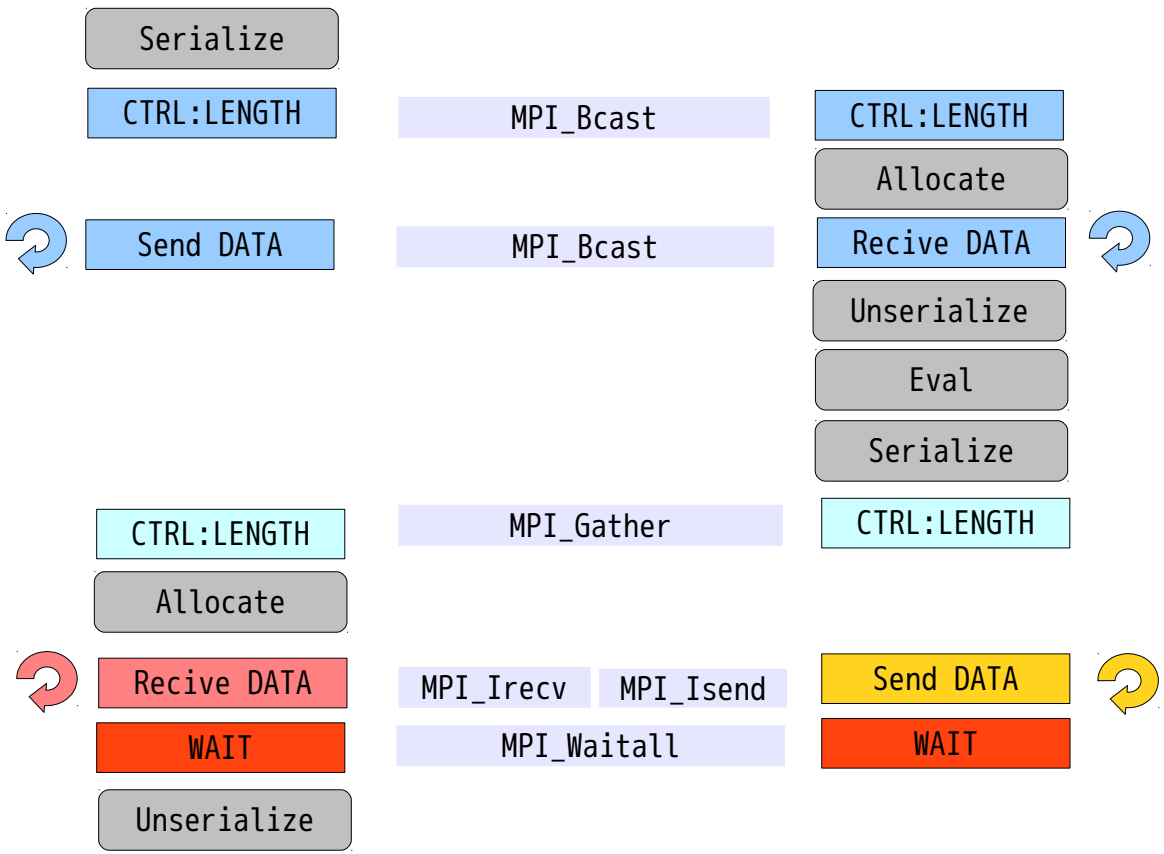
- Apply type functions
  - Rhpc_lapply (∼ snow::clusterApply)
  - Rhpc_lapplyLB (∼ snow::clusterApplyLB)
- Function to set up random number generators
  - Rhpc_setupRNG (∼ snow::clusterSetupRNGstream)
- Miscellaneous functions
  - Rhpc_worker_noback
    (Experimental I/F for calling foreign SPMD program)
  - Rhpc_serialize, Rhpc_unserialize
    (C functions for serialization and unserialization)

# Rhpc_worker_call (1)

## Rhpc_worker_call(cl, FUN, ...)

- cl
  pointer to communicator
- FUN
  Function name or string (string expresses function name)
  Distributed by **collective communication**
- ... (argument)
  Distributed by **collective communication**

# Rhpc_worker_call (2)

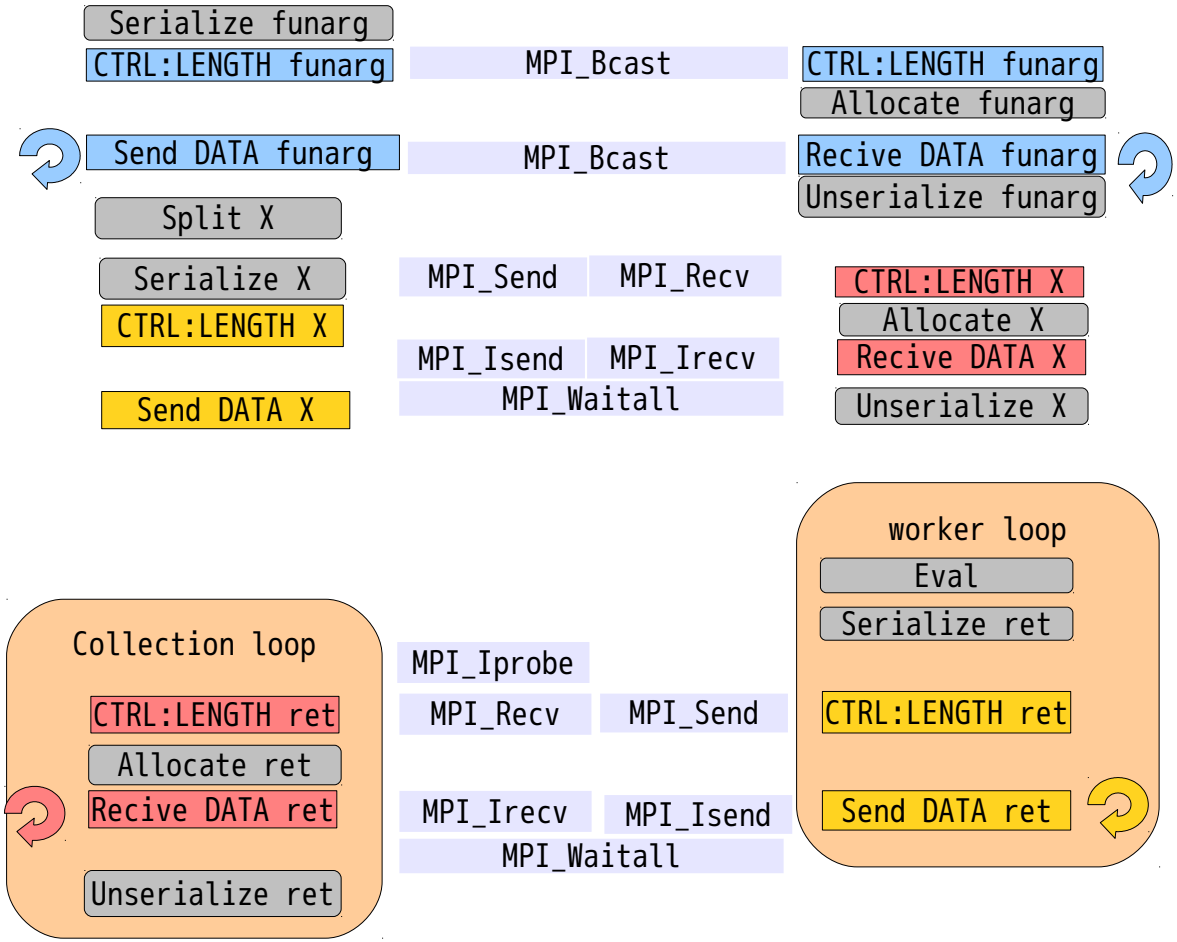


# Rhpc_lapply (1)

## Rhpc_lapply(cl, x, FUN, ...)

- cl
  pointer to communicator
- x
  vector or list.
  Divided into smaller vectors according to the number of workers, and distributed to workers when the function is first executed.
  **One-sided communication is used asynchronously**.
- FUN
  Function name or string (string expresses function name)
  Distributed by **collective communication** at first, then they **are not sent again**
- ... (argument)
  Distributed by **collective communication** at first, then they **are not sent again**.

# Rhpc_lapply (2)

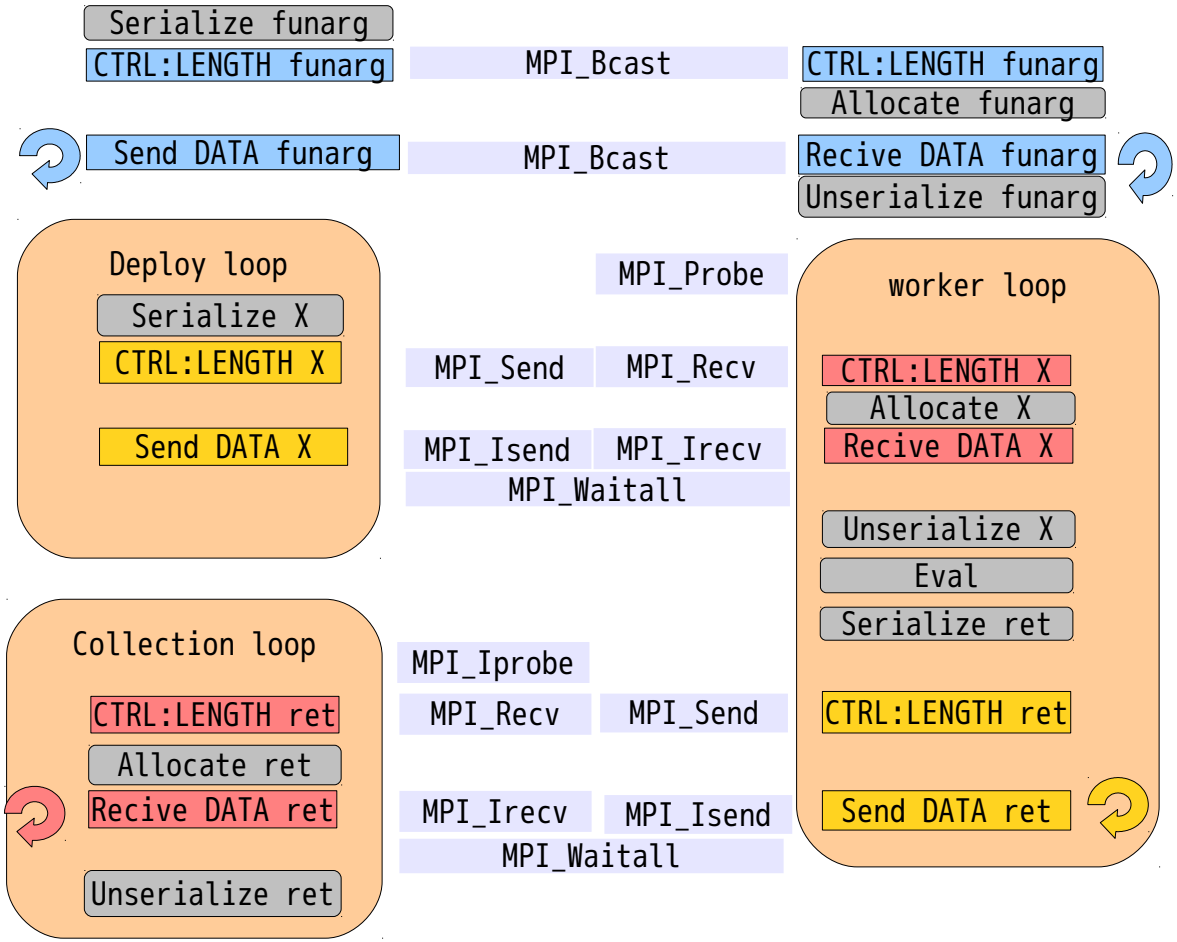


# Rhpc_lapplyLB (1)

## Rhpc_lapplyLB(cl, x, FUN, ...)

- cl
  pointer to communicator
- x
  vector or list.
  Distributed to workers when the function is executed. **One-sided communication is used asynchronously**.
- FUN
  Function name or string (string expresses function name)
  Distributed by **collective communication** at first, then they **are not sent again**.
- ... (argument)
  Distributed by **collective communication** at first, then they **are not sent again**.

# Rhpc_lapplyLB (2)



# Huge data example: snow (SOCK)

As snow(SOCK) utilizes pipe to serialize and unserialize data, it can handle huge data. However, socket mechanism has difficulty to control processes on many nodes by many users in supercomputer environment

## use snow(SOCK)::clusterExport in one worker

```
> library(snow)
> cl<-makeCluster(1,type="SOCK")
> set.seed(123)
> N<-17e3
> M<-matrix(runif(N^2),N,N)
> sum(M)
[1] 144501466
> system.time(clusterExport(cl,"M"))
   user  system elapsed
  4.213   1.580   8.228
> f<-function()sum(M)
> clusterCall(cl,f)
[[1]]
[1] 144501466
```

# Huge data example: snow(MPI)

At present, Rmpi cannot handle data more than 2GB, because the argument of MPI_send etc. should be int size.

## use snow(MPI)::clusterExport in one worker

```
> library(snow)
> cl<-makeCluster(1,type="MPI")
> set.seed(123)
> N<-17e3
> M<-matrix(runif(N^2),N,N)
> sum(M)
[1] 144501466
> system.time(clusterExport(cl,"M"))
Error in mpi.send(x = serialize(obj, NULL), type = 4, dest = dest, tag = tag,  :
  long vectors not supported yet: memory.c:3100
Calls: system.time ... sendData.MPInode -> mpi.send.Robj -> mpi.send -> .Call
```

# Huge data example: Rhpc

Rhpc can handle huge data, because it divides huge data into appropriate size and uses MPI repeatedly.

## use Rhpc::Rhpc_Export in one worker

```
> library(Rhpc)
> Rhpc_initialize()
> cl<-Rhpc_getHandle()
> set.seed(123)
> N<-17e3
> M<-matrix(runif(N^2),N,N)
> sum(M)
[1] 144501466
> system.time(Rhpc_Export(cl,"M"))
   user  system elapsed
  9.241   1.700  10.972
> f<-function()sum(M)
> Rhpc_worker_call(cl,f)
[[1]]
[1] 144501466
> Rhpc_finalize()
```

# Many workers example (1): snow(MPI)

As clusterCall of snow starts workers sequentially, it becomes slow when the number of workers increases.

## use snow(MPI)::clusterExport in 63 workers

```
> library(Rmpi)
> library(snow)
> cl<-makeMPIcluster()
> set.seed(123)
> N<-4e3
> length(cl)
[1] 63
> M<-matrix(runif(N^2),N,N)
> system.time(clusterExport(cl,"M"))
   user  system elapsed
 26.715  10.903  37.761
> f<-function()sum(M)
> all.equal(rep(sum(M),length(cl)),unlist(clusterCall(cl,f)))
[1] TRUE
> stopCluster(cl)
```

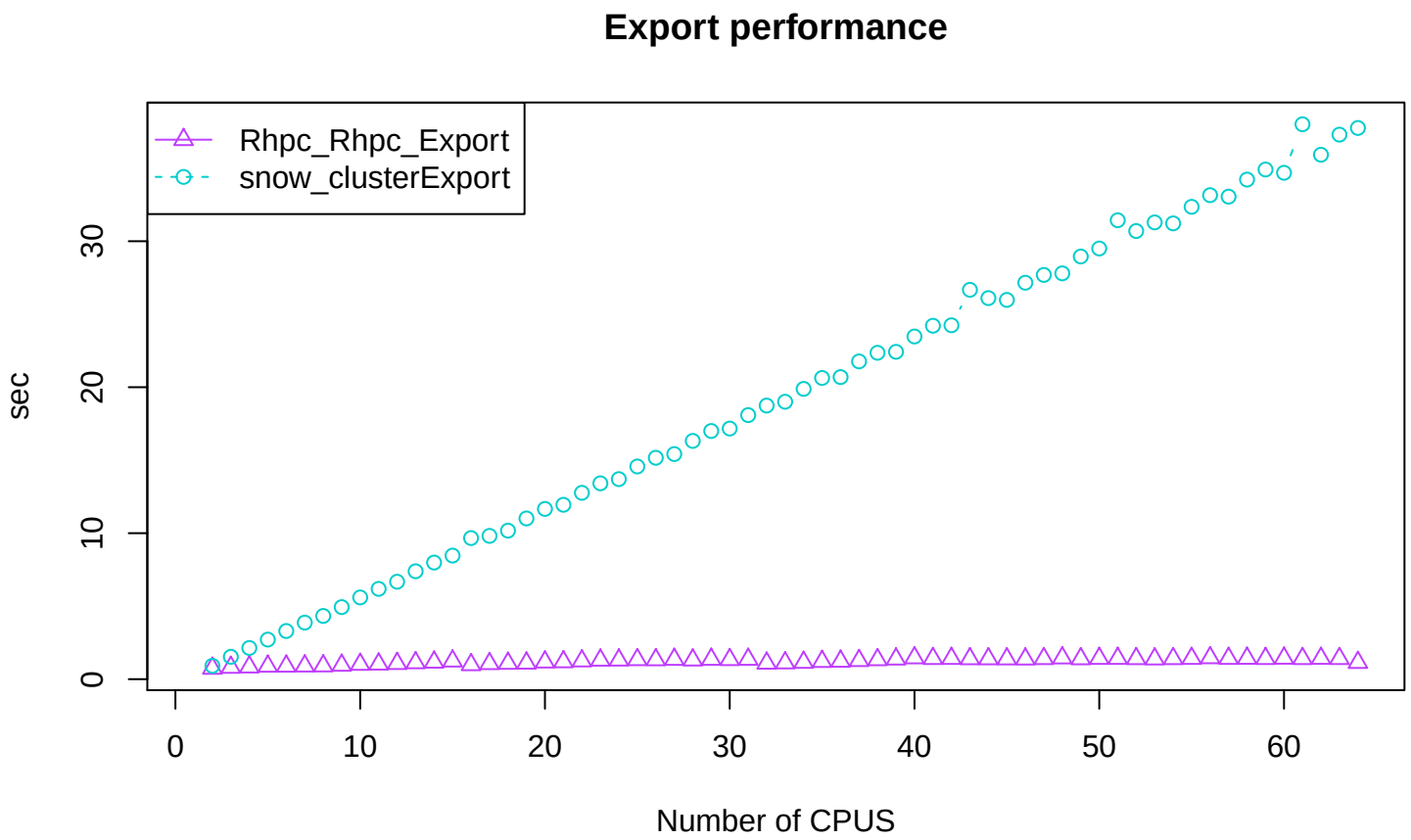
# Many workers example (1): Rhpc

As Rhpc uses collective communication by MPI_Bcast, data transportation to workers is still fast even when the number of workers increases.

## use Rhpc::Rhpc_Export in 63 workers

```
> library(Rhpc)
Loading required package: rlecuyer
> Rhpc_initialize()
> cl<-Rhpc_getHandle()
Detected communication size 64
> set.seed(123)
> N<-4e3
> Rhpc_numberOfWorker(cl)
[1] 63
> M<-matrix(runif(N^2),N,N)
> system.time(Rhpc_Export(cl,"M"))
   user  system elapsed
  1.012   0.116   1.139
> f<-function()sum(M)
> all.equal(rep(sum(M),Rhpc_numberOfWorker(cl)),unlist(Rhpc_worker_call(cl,f)))
[1] TRUE
> Rhpc_finalize()
```

# Many workers example (1): Rhpc and snow(MPI)



# Many workers example (2a): snow(MPI)

As the main parts of snow and Rmpi are written in R language, they are rather slow.

## use snow(MPI)::clusterApply in 63 workers

```
> library(Rmpi)
> library(snow)
> cl<-makeMPIcluster()
> system.time(ans<-clusterApply(cl,1:10000,sqrt))
   user  system elapsed
  1.423   0.005   1.429
> all.equal(sqrt(1:10000),unlist(ans))
[1] TRUE
> stopCluster(cl)
```

# Many workers example (2b): snow(MPI)

As the load balancing function is written in R language, it becomes slow according to the number of parallel workers.

## use snow(MPI)::clusterApplyLB in 63 workers

```
> library(Rmpi)
> library(snow)
> cl<-makeMPIcluster()
> system.time(ans<-clusterApplyLB(cl,1:10000,sqrt))
   user  system elapsed
  4.395   0.003   4.413
> all.equal(sqrt(1:10000),unlist(ans))
[1] TRUE
> stopCluster(cl)
```

# Many workers example (2a): Rhpc

As the main part of Rhpc is written in C language, it is efficient.

## use Rhpc::Rhpc_lapply in 63 workers

```
> library(Rhpc)
Loading required package: rlecuyer
> Rhpc_initialize()
> cl<-Rhpc_getHandle()
Detected communication size 64
> system.time(ans<-Rhpc_lapply(cl, 1:10000, sqrt))
   user  system elapsed
  0.045   0.001   0.046
> all.equal(sqrt(1:10000),unlist(ans))
[1] TRUE
> Rhpc_finalize()
```

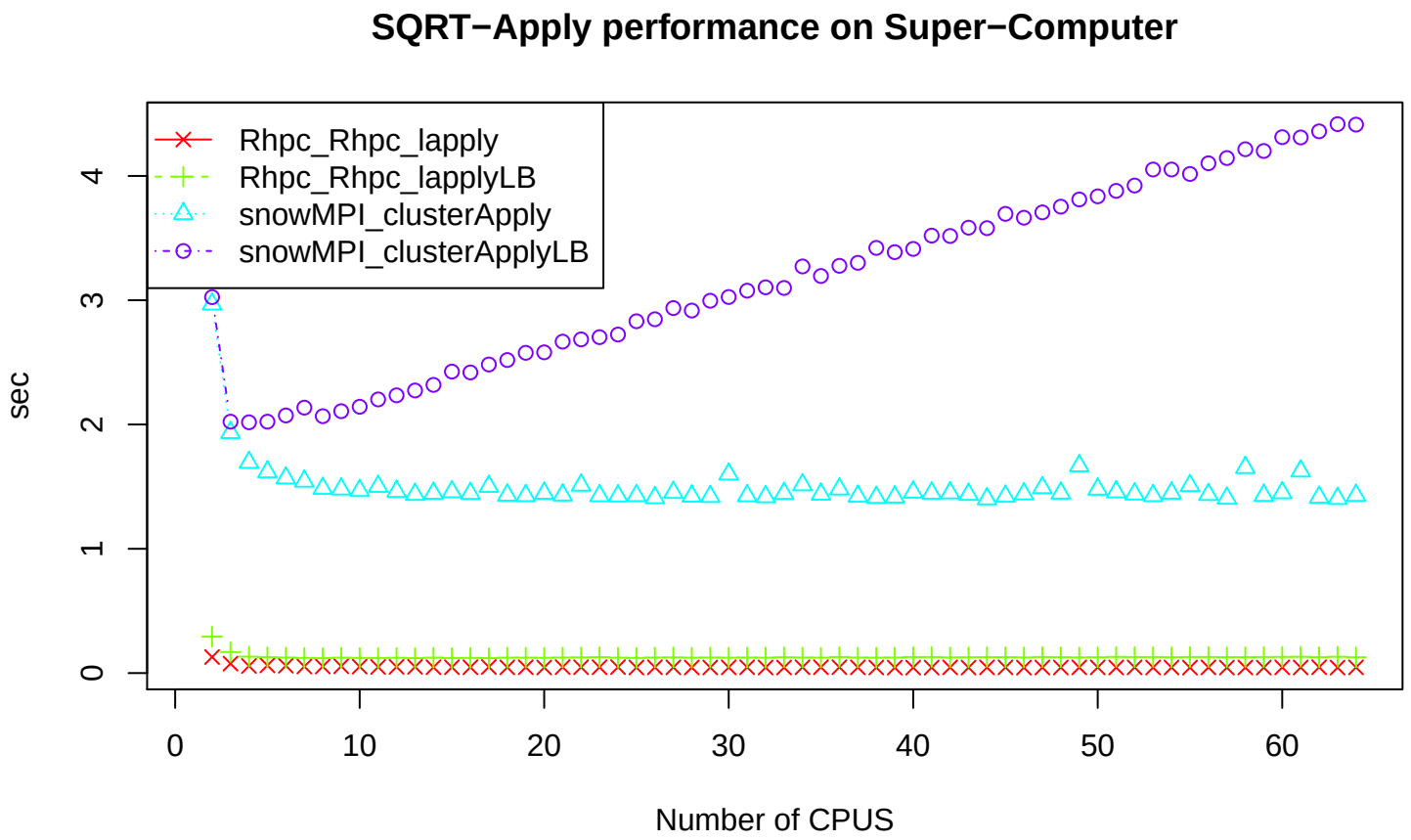
# Many workers example (2b): Rhpc

Rhpc_lapplyLB sends the argument x to an available worker. So it produces a little delay.

## use Rhpc::Rhpc_lapplyLB in 63 workers

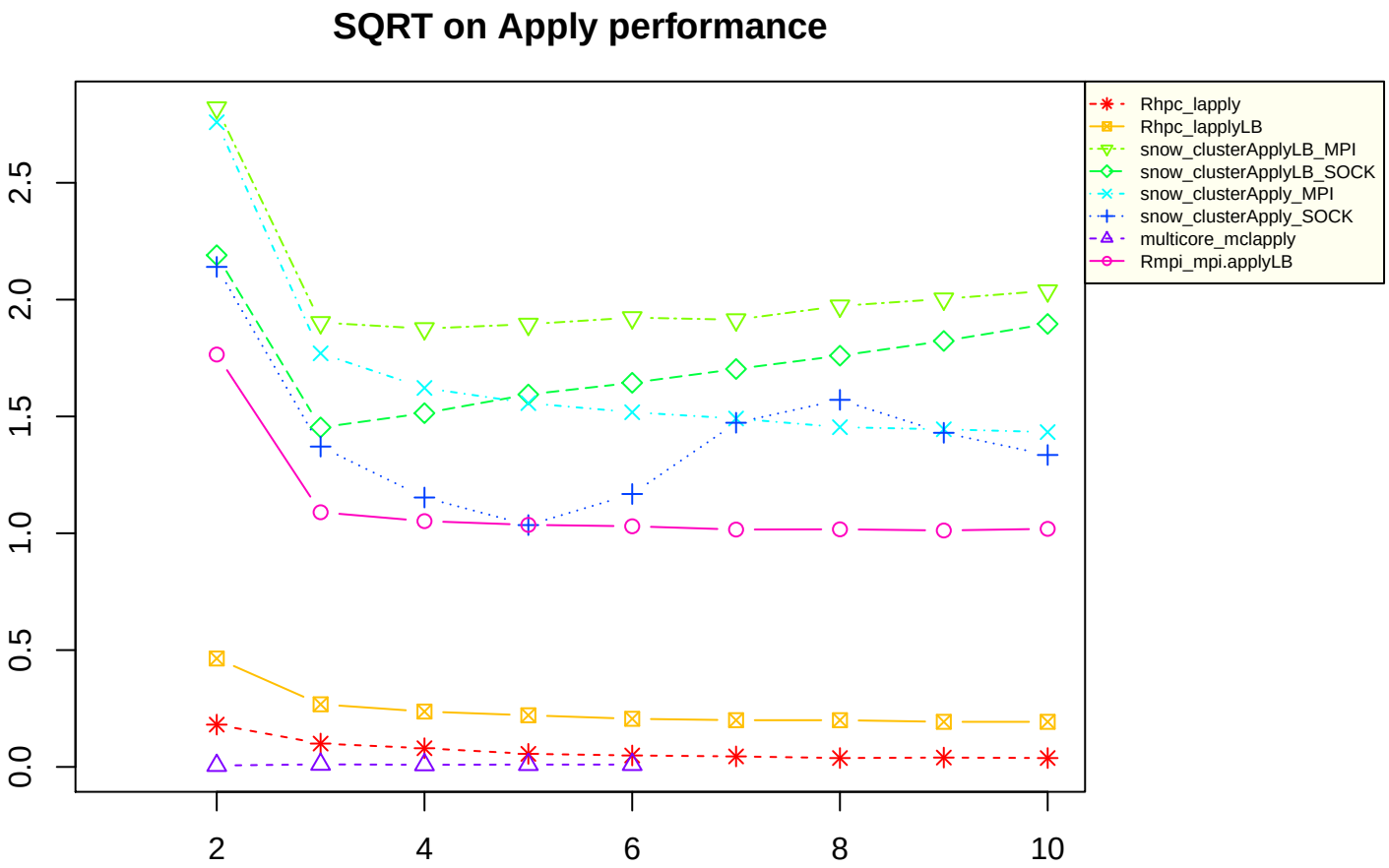
```
> library(Rhpc)
Loading required package: rlecuyer
> Rhpc_initialize()
> cl<-Rhpc_getHandle()
Detected communication size 64
> system.time(ans<-Rhpc_lapplyLB(cl, 1:10000, sqrt))
   user  system elapsed
  0.125   0.001   0.127
> all.equal(sqrt(1:10000),unlist(ans))
[1] TRUE
> Rhpc_finalize()
```

# Many workers example (2): Rhpc and snow(MPI)



# Several parallel apply functions

At present, `Rhpc` is a little slower than `multicore` on one CPU.



# Interface to foreign programs using MPI

C and/or Fortran programs using MPI adopt SPMD programming, in which Master(rank0) communicates with Workers(their ranks are more than 1) using communicators. Therefore, when `Rhpc`, which uses MPI, call such programs, exchanging MPI communicators is required. `Rhpc` provides Rhpc_worker_noback function to support it.

`Rhpc` sets global variable _options_ (see options function):

- `Rhpc.mpi.f.comm`
  `Comuunicator for Fortran (R type:  integer)`
- `Rhpc.mpi.c.comm`
  `Communicator for C (R type:  external pointer)`
- `Rhpc.mpi.procs`
  `Communication size in MPI`
- `Rhpc.mpi.rank`
  `Rank in MPI`

# Using '.Fortran', '.C' and '.Call' in R

Foreign programs (Fortran or C) using MPI are called in R

```
1  mpipif<-function(n)
2  {
3    ## Exported functions get values by getOption()
4    ## when they run on workers
5    out<-.Fortran("mpipif",
6                  comm=getOption("Rhpc.mpi.f.comm"),
7                  n=as.integer(n),
8                  outpi=as.double(0))
9    out$outpi
10 }
```

```
1  mpipic<-function(n)
2  {
3    ## Exported functions get values by getOption()
4    ## when they run on workers
5    out<-.C("mpipic",
6            comm=getOption("Rhpc.mpi.f.comm"),
7            n=as.integer(n),
8            outpi=as.double(0))
9    out$outpi
10 }
```

```
1  mpipicall<-function(n)
2  {
3    ## Exported functions get values by getOption()
4    ## when they run on workers
5    out<-.Call("mpipicall",
6               comm=getOption("Rhpc.mpi.c.comm"),
7               n=as.integer(n))
8    out
9  }
```

Note that '.C' cannot receive external pointer and integer communicator for Fortran should be used. Thus . '.Call' is preferable for C programs.

# Changing MPI Fortran code for '.Fortran' in R.

```
      program main                                  |       subroutine mpipif(mpi_comm,n,outpi)
      include "mpif.h"                                      include "mpif.h"
      double precision  mypi, sumpi                         double precision  mypi, sumpi
      double precision  h, sum, x, f, a                     double precision  h, sum, x, f, a
      double precision  pi                                  double precision  pi
      parameter (pi=3.14159265358979323846)                 parameter (pi=3.14159265358979323846)
      integer n, rank, procs, i, ierr                       integer n, rank, procs, i, ierr
      character*16 argv                             |       integer mpi_comm
      integer argc                                  |       double precision  outpi
      f(a) = 4.d0 / (1.d0 + a*a)                            f(a) = 4.d0 / (1.d0 + a*a)
      argc = COMMAND_ARGUMENT_COUNT()               <
      n=0                                           <
      if (argc .ge. 1) then                         <
        call getarg(1, argv)                        <
        read(argv,*) n                              <
      endif                                         <
      call MPI_INIT(ierr)                           <
c     COMM                                                c     COMM
      call MPI_COMM_RANK(MPI_COMM_WORLD,            |       call MPI_COMM_RANK(mpi_comm,
     &                   rank, ierr)                        &                   rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,            |       call MPI_COMM_SIZE(mpi_comm,
     &                   procs, ierr)                       &                   procs, ierr)
      call MPI_BCAST(n,1,MPI_INTEGER,0,                     call MPI_BCAST(n,1,MPI_INTEGER,0,
     &               MPI_COMM_WORLD,ierr)           |       &               mpi_comm,ierr)
      if ( n .le. 0 ) goto 30                               if ( n .le. 0 ) goto 30
      h = 1.0d0/n                                           h = 1.0d0/n
      sum  = 0.0d0                                          sum  = 0.0d0
      do 20 i = rank+1, n, procs                            do 20 i = rank+1, n, procs
         x = h * (dble(i) - 0.5d0)                             x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)                                      sum = sum + f(x)
20    continue                                              20    continue
      mypi = h * sum                                        mypi = h * sum
      call MPI_REDUCE(mypi,sumpi,1,                         call MPI_REDUCE(mypi,sumpi,1,
     &                MPI_DOUBLE_PRECISION,                 &                MPI_DOUBLE_PRECISION,
     &                MPI_SUM,0,                            &                MPI_SUM,0,
     &                MPI_COMM_WORLD,ierr)          |       &                mpi_comm,ierr)
      if (rank .eq. 0) then                         |       outpi=sumpi
         print *, 'pi = ', sumpi                    |   30  continue
      endif                                         |       return
30    call MPI_FINALIZE(ierr)                       <
      stop                                          <
      end                                                   end
```

# Changing MPI C code for '.C' in R.

```
#include "mpi.h"                                  #include "mpi.h"
#include <stdio.h>                                #include <stdio.h>
#include <math.h>                                 #include <math.h>
                                                >  #include <R.h>
                                                >  #include <Rinternals.h>
int main( int argc, char *argv[] )              |  int mpipic( int *comm, int *N, double *outpi )
{                                                  {
                                                >    MPI_Comm mpi_comm;
  int n=0, rank, procs, i;                         int n=0, rank, procs, i;
  double mypi, pi, h, sum, x;                      double mypi, pi, h, sum, x;
  if ( argc >= 2){                              |    mpi_comm = MPI_Comm_f2c(*comm);
    n = atoi(argv[1]);                          |    n = *N;
  }                                             <
                                                <
  MPI_Init(&argc,&argv);                           // COMM
  // COMM                                       |    MPI_Comm_size(mpi_comm,      &procs);
  MPI_Comm_size(MPI_COMM_WORLD,&procs);         |    MPI_Comm_rank(mpi_comm,      &rank);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);          |    MPI_Bcast(&n, 1, MPI_INT, 0, mpi_comm);
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);    h   = 1.0 / (double) n;
  h   = 1.0 / (double) n;                          sum = 0.0;
  sum = 0.0;                                       for (i = rank + 1; i <= n; i += procs) {
  for (i = rank + 1; i <= n; i += procs) {           x = h * ((double)i - 0.5);
    x = h * ((double)i - 0.5);                       sum += (4.0 / (1.0 + x*x));
    sum += (4.0 / (1.0 + x*x));                    }
  }                                                mypi = h * sum;
  mypi = h * sum;                                  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);                   |             mpi_comm);
  if (rank == 0)                                |    *outpi=pi;
    printf("pi = %.16f\n", pi);                 <
  MPI_Finalize();                               <
  return(0);                                       return(0);
}                                                }
```

# Changing MPI C code for '.Call' in R.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[] )
{


  int n=0, rank, procs, i;
  double mypi, pi, h, sum, x;
  if ( argc >= 2){
    n = atoi(argv[1]);
  }
  MPI_Init(&argc,&argv);
  // COMM
  MPI_Comm_size(MPI_COMM_WORLD,&procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = rank + 1; i <= n; i += procs) {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (rank == 0)
    printf("pi = %.16f\n", pi);
  MPI_Finalize();
  return(0);
}
```

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
> #include <R.h>
> #include <Rinternals.h>
| SEXP    mpipicall(SEXP comm, SEXP N)
  {
>   MPI_Comm mpi_comm;
>   SEXP ret;
    int n=0, rank, procs, i;
    double mypi, pi, h, sum, x;
|   mpi_comm = *((MPI_Comm*)R_ExternalPtrAddr(comm));
|   PROTECT(ret=allocVector(REALSXP,1));
|   n = INTEGER(N)[0];
<
    // COMM
|   MPI_Comm_size(mpi_comm,      &procs);
|   MPI_Comm_rank(mpi_comm,      &rank);
|   MPI_Bcast(&n, 1, MPI_INT, 0, mpi_comm      );
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += procs) {
      x = h * ((double)i - 0.5);
      sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
|              mpi_comm      );
|   REAL(ret)[0]=pi;
|   UNPROTECT(1);
|   return(ret);
<
  }
```

# Call foreign MPI program from R

```
1  source("mpipicall.R")
2  source("mpipic.R")
3  source("mpipif.R")
4
5  library(Rhpc)
6  Rhpc_initialize()
7  cl<-Rhpc_getHandle(4)
8
9  n<-100
10
11 ## Load shared library
12 Rhpc_worker_call(cl,dyn.load,"pi.so"); dyn.load("pi.so")
13
14 ## Rhpc_worker_noback calls a function, but does not
15 ## get any result.
16 ## Workers should be started faster than a master.
17 Rhpc_worker_noback(cl,mpipicall,n); mpipicall(n)
18 Rhpc_worker_noback(cl,mpipic,n); mpipic(n)
19 Rhpc_worker_noback(cl,mpipif,n); mpipif(n)
20
21 Rhpc_finalize()
```

# Old serialization

At first, we realized serialization by calling R interpreter from C program as below. It was slow.

## one-time call serialize

```
SEXP boo = LCONS(install("serialize"),
                 CONS(args,
                      CONS(R_NilValue,
                           CONS(ScalarLogical(FALSE),
                                CONS(R_NilValue,
                                     CONS(R_NilValue,
                                          R_NilValue))))));
boo = LCONS(install(".Internal"), CONS(boo, R_NilValue));
SEXP ret = R_tryEval(boo, R_GlobalEnv, &errorOccurred));
```

# New serialization

We extract code for serialization from R source code. It is faster about 5 times than the old serialization program. This idea is used to produce `RApiSerialize package by Dirk Eddelbuettel.`

**new call serialize**

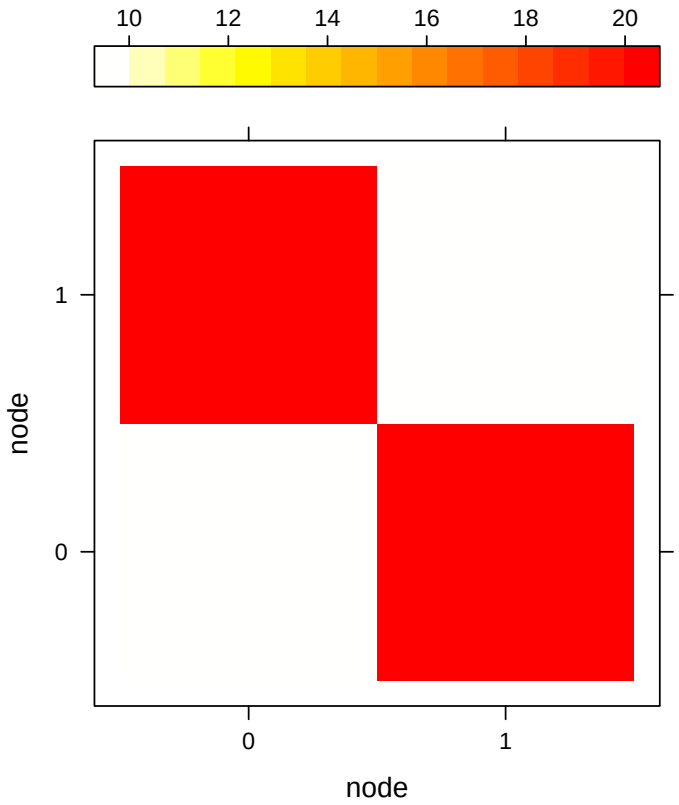
```
out=Rhpc_serialize(args);
```

# Outline

# NUMA distances

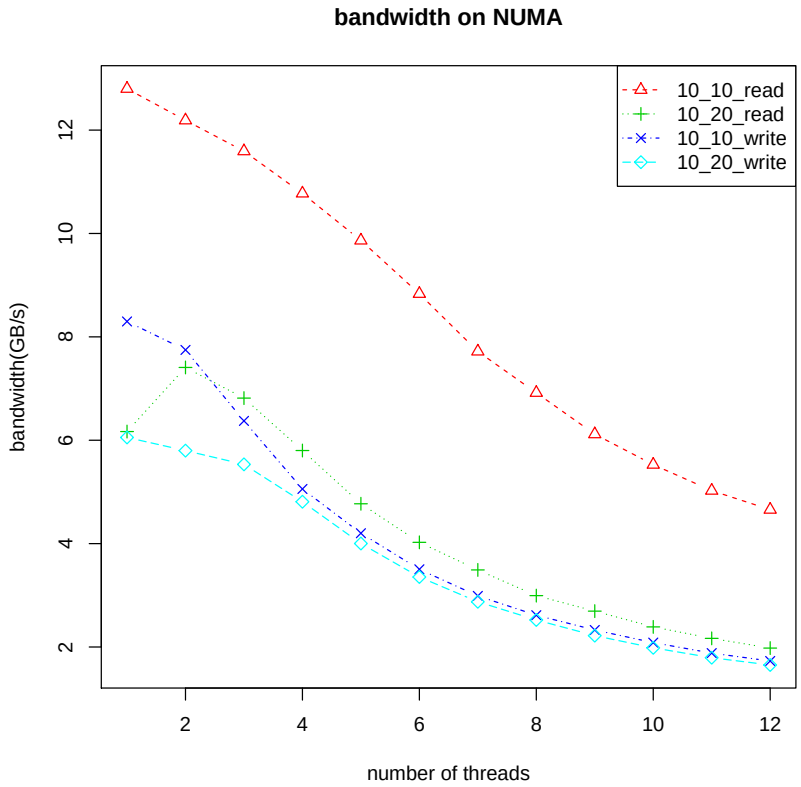
Almost all present multi-CPU and multi-core computers are NUMA (Non-Uniform Memory Access) shared memory system.

NUMA distances between two nodes (= CPUs) are very different between CPUs. This affects the memory bandwidth.



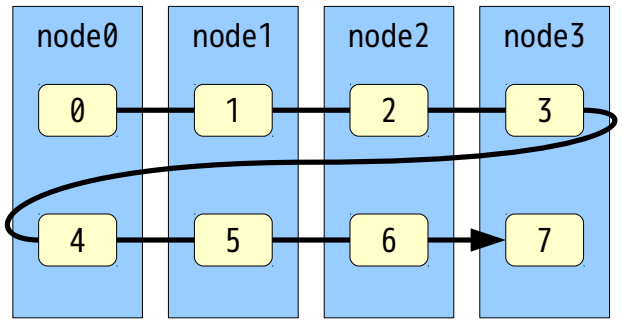
# Memory bandwidth in NUMA

We checked the memory bandwidth according to values of NUMA distance. Memory bandwidth for reading is significantly small in THIS CASE.



# GotoBLAS or OpenBLAS on NUMA system

GotoBLAS or OpenBLAS are indispensable for high performance (parallel) computing. They try to assign threads to cores of each nodes in turn for using CPU cache efficiently in the default setting.




If we use MPI and BLAS together, we should be careful about their CPU affinity.

# NUMA and MPI

- Although MPI has been developed mainly for distributed memory system, it is also useful for present NUMA shared memory system.
- We recommend to distribute resources in NUMA system by using bind functions of OpenMPI and our `RhpcBLASctl` package.

# Package RhpcBLASctl

- Functions to control the number of threads on BLAS, MKL, ACML and GotoBLAS(or OpenBLAS)
  - `blas_get_num_procs()`
  - `blas_set_num_threads(threads)`
- Functions to control the number of threads on OpenMP
  - `omp_get_num_procs()`
  - `omp_get_max_threads()`
  - `omp_set_num_threads(threads)`
- Functions to get the number of physical and logical cores
  - `get_num_cores() # physical cores`
  - `get_num_procs() # logical cores`

# Simple example of RhpcBLASctl

Number of threads for parallel BLAS can be controlled to reserve cores.

## pararrel BLAS control

```
> library(RhpcBLASctl)
> A<-matrix(runif(3e3*3e3),3e3,3e3)
> blas_set_num_threads(1)
> blas_get_num_procs()
[1] 1
> system.time(A%*%A)
   user  system elapsed
  3.992   0.012   4.003
> get_num_cores()
[1] 4
> blas_set_num_threads(get_num_cores())
> system.time(A%*%A)
   user  system elapsed
  4.332   0.048   1.234
```

# Example to use MPI and parallel BLAS: R code

## emm.R: Performing 8 parallel DGEMM in MKL

```
library(RhpcBLASctl)
library(Rhpc)
Rhpc_initialize()
cl<-Rhpc_getHandle()
dummy<-Rhpc_EvalQ(cl,library(RhpcBLASctl))
Rhpc_worker_call(cl,blas_set_num_threads,8)
unlist(Rhpc_worker_call(cl,blas_get_num_procs))
f<-function(M) {
   res<-sapply(1:5,function(x)x;system.time(M%*%M)[[3]])
   sort(res)
}
N<-3e3
M<-matrix(runif(N^2),N,N)
system.time(ans<-Rhpc_worker_call(cl,f,M))
(unlist(ans))
Rhpc_finalize()
```

# Example to use MPI and parallel BLAS: JOB script

## gemm_s.sh: JOB script

```
#!/bin/bash
#PBS -q ice-s
#PBS -l select=1:ncpus=24:mem=120gb
#PBS -N gemm_s
#PBS -o gemm_s.out
#PBS -e gemm_s.err
#PBS -V
cd ${PBS_O_WORKDIR}
OMP_NUM_THREADS=1
OMP_DYNAMIC=FALSE
MKL_DYNAMIC=FALSE
export MKL_DYNAMIC
export OMP_DYNAMIC
export OMP_NUM_THREADS
mpirun --report-bindings --bind-to-socket --bysocket -np 3 \
       /home/nakama/R/x86_64-unknown-linux-gnu-library/3.1/Rhpc/Rhpc \
       CMD BATCH --no-save gemm.R gemm_s.Rout
```

Note that

- MKL_DINAMIC=FALSE
  specifies the number of threads manually
- mpirun –bind-to-socket option
  specifies binding processes to processor sockets

# Example to use MPI and parallel BLAS: Report from openmpi

```
[r5i2n7:06743] MCW rank 0 bound to socket 0[core 0-11]: [B B B B B B B B B B B B][. . . . . . . . . . . .]
[r5i2n7:06743] MCW rank 1 bound to socket 1[core 0-11]: [. . . . . . . . . . . .][B B B B B B B B B B B B]
[r5i2n7:06743] MCW rank 2 bound to socket 0[core 0-11]: [B B B B B B B B B B B B][. . . . . . . . . . . .]
```

- rank0 master thread works on some core of socket 0
- rank1 worker threads (8 DGEMMs) works on some cores of socket 1
- rank2 worker threads (8 DGEMMs) works on some cores of socket 0

# Example to use MPI and parallel BLAS: Result

gemm_s.Rout:

```
> library(RhpcBLASctl)
> library(Rhpc)
> Rhpc_initialize()
reload mpi library /home/nakama/lib64/libmpi.so.1
rank 0/ 3(0) : r5i2n7 : 6750
> cl<-Rhpc_getHandle()
Detected communication size 3
> dummy<-Rhpc_EvalQ(cl,library(RhpcBLASctl))
> dummy<-Rhpc_worker_call(cl,blas_set_num_threads,8)
> unlist(Rhpc_worker_call(cl,blas_get_num_procs))
[1] 8 8
> f<-function(M) {
+ res<-sapply(1:5,function(x){x;system.time(M%*%M)
       [[3]]})
+ sort(res)
+ }
> N<-3e3
> M<-matrix(runif(N*N),N,N)
> system.time(ans<-Rhpc_worker_call(cl,f,M))
   user system elapsed
  0.828 2.008 2.838
> (unlist(ans))
 [1] 0.375 0.376 0.376 0.376 0.986 0.386 0.387 0.387
        0.387 0.995
> Rhpc_finalize()
```

- We see elapsed times to check whether MKL works by using 8 threads or not
- If we use –bind-to-core option of mpirun, just 1 thread is available

# Example to use MPI and parallel BLAS: Performance of DGEMM

```
> ## logical peek
> cpuclock <- 2.7e9 # 2.7GHz
> numthreads <- 8
> cycle <- 4 # SIMD=2 AVX=4 AVX2=8
> (logicalpeek <- 2*cycle*cpuclock*numthreads) # GFLOPS
[1] 1.728e+11
>
> ## computational complexity
> N <- 3000
> (FLO <- 2*N^3) # O(N^3)
[1] 5.4e+10
>
> ## effective energy efficiency
> realsec <- 0.387
> (FLO/realsec/logicalpeek)
[1] 0.8074935
>
> proc.time()
   user system elapsed
  0.128 0.016 0.142
```

# Outline

# Concluding remarks

- In the ISM, R has been used on supercomputers for big data analysis and/or large scale simulations.
- Good parallel technology is required for R on supercomputers.
- `Rhpc` is an attempt for this purpose. `Rhpc` depends heavily on MPI implementation and NUMA structure.
- `Rhpc` is still under development and we are tuning it to get more efficiency.