PARALLEL R WITH MPI

Jacalyn Huband
UVA Research Computing

MPI

- MPI (i.e., Message Passing Interface) controls communication among nodes on a cluster

- It is a standard protocol for communication on a high-performance cluster (HPC)

- With MPI, we can distribute the work of code across 2 or more nodes

Using Multiple Nodes

- Why use multiple nodes?
  - Access to many more cores
    - Able to use more cores than on a single node
    - Possibly save computational time
  - Access to more memory

Goals for Today

- Learn how to submit an MPI job on Rivanna;
- Learn how to use a subset of the MPI function available in R;
- Practice running MPI jobs on Rivanna

SUBMITTING AN MPI JOB ON RIVANNA

Submitting MPI jobs on Rivanna

- The best way to run MPI jobs on Rivanna is through a SLURM script.
- If you are not familiar with creating and submitting SLURM scripts, we have the basics posted on our website: https://www.rc.virginia.edu/userinfo/rivanna/slurm/

We will see example SLURM scripts on subsequent slides.

University of Virginia | Research Computing

OpenMPI on Rivanna

- Because MPI is a protocol, we need a software application to implement the protocol.
- There are different applications of MPI on Rivanna (e.g., OpenMPI and intelMPI)
- We will use OpenMPI:

```
module load gcc/7.1.0 openmpi/3.1.4
```

OR

```
module load goolf/7.1.0_3.1.4
```

The name "goolf" stands for **g**cc, **o**penmpi, **o**penblas, **l**apack, scalapack and **f**ftw

To use MPI:

- You must use the parallel partition.
- You must request more than one Node.
- You must use `srun` to launch the job.

Sample SLURM script:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 5
#SBATCH -n 10
#SBATCH -t 00:03:00

module purge
module load goolf/7.1.0_3.1.4

srun hostname
```

University of Virginia | Research Computing

Hands-on Activity #0:

- Log onto Rivanna and run this SLURM script:
- Check the output. What do you see?

- Remove the word `srun` from the executable line and rerun the script.
- What is different about the output?

All codes used in this worksop are available on Rivanna.
To copy them into your account, type:
`cp -r /project/rivanna-training/R_with_MPI .`

Sample SLURM script:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 5
#SBATCH -n 10
#SBATCH -t 00:03:00

module purge
module load goolf/7.1.0_3.1.4

srun hostname
```

MPI Basic Concepts: Multiple Copies

- The basic idea behind MPI is that multiple copies of your code are running on different cores and probably on different nodes.

- You, as the programmer, must design your code so that the copies know their individual responsibilities and when to communicate results with each other.

MPI Basic Concepts: Rank

- Each copy of your code is called a *task*.
- Each task will be assigned a number, called its *rank*.
  - The ranks will be 0, 1, …, (N-1), where N is the total number of copies of your program (as defined by `ntasks` or `n` in SLURM).
  - With MPI functions, each copy can know its rank and the total number of copies running.
- The rank can be used to assign responsibilities in the code.

MPI and R

- We will use the pbdMPI package with R.
- All programs that use pbdMPI should start with an `init()` function and *must* end with a `finalize()` function. These functions set up and take down the MPI communications among the nodes.
- The function `comm.rank()` will return the rank of the given copy and `comm.size()` will return the number of copies that are running.
- Printing will require either `comm.print()` or `comm.cat()`.

HELLO WORLD EXAMPLE

Note: MPI codes will not run inside of Rstudio.

MPI Example #1

ex1_helloWorld.R:

```
library(pbdMPI, quiet=TRUE)
init()

#   Without specifying the
#   rank, only the process
#   with rank 0 will print

comm.cat("\nHello World!\n")

finalize()
```

ex1.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 3
#SBATCH -n 6
#SBATCH -t 00:03:00

module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex1_helloWorld.R
```

Submit this script by typing

```
    sbatch ex1.slurm
```

in a terminal window.

MPI Example #1

ex1_helloWorld.R:

```
library(pbdMPI, quiet=TRUE)
init()

#   Without specifying the
#   rank, only the process
#   with rank 0 will print

comm.cat("\nHello World!\n")

finalize()
```

ex1.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 3
#SBATCH -n 6
#SBATCH -t 00:03:00

module purge
module load goolf/7.1.0_3.1.4 R

ipt ex1_Hello.R
```

There will be 6 copies running on 3 nodes. SLURM will decide how many copies should be on each node.

Expected Results:

```
        COMM.RANK = 0

        Hello World!
```

Hands-on Activity #1

Background Information:

When we don't tell the `comm.cat` function who should print, it will default to the task of rank 0 only.

To tell it which tasks should print, we can include in the function call a list of ranks. For example:

```
comm.cat("\nHello World!\n",rank.print=c(0,3))
```

There may be times when we want all ranks to print their data. In this case, we can use an all.rank option. For example:

```
comm.cat("\nHello World!\n",all.rank=TRUE)
```

Activities:

1. Modify the print command so that tasks 2 and 4 print the message.
2. Modify the print command so that all task print the message.

BARRIER

Allowing tasks to synchronize

Barrier Concepts

- There may be times when you need all copies of your code to finish a computation before the any of them proceed to the next step.
- We can control this with a `barrier()` function.
- The `barrier()` function is like a road block – it prevents any of the tasks for moving forward until all of the tasks have caught up.
- When needed, it is essential; but it should be used judiciously to prevent unnecessary slowdowns of the code

MPI Example #2

ex2_barrier.R:

```
library(pbdMPI, quiet=TRUE)
init()
myRank <- comm.rank()

# Computation that varies in amount of time
z <- 1000*(1 + myRank)
if (myRank > 0){
   Sys.sleep(myRank)
   comm.print(z, rank.print=myRank, quiet=TRUE)
}

barrier()   #  The barrier ensures that all
            # work is complete before the
            # "All Done" would be reported
comm.print(“All Done”)
finalize()
```

ex2.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex2_barrier.R
```

MPI Example #2

ex2_barrier.R:

```
library(pbdMPI, quiet=TRUE)
init()
myRank <- comm.rank()

# Computation that varies in amount of time
z <- 1000*(1 + myRank)
if (myRank > 0){
   Sys.sleep(myRank)
   comm.print(z, rank.print=my
}

barrier()  #  The barrier ensu
           # work is complete 
           # "All Done" would 
comm.print("All Done")
finalize()
```

ex2.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
 SBATCH -t 00:03:00


odule purge
odule load goolf/7.1.0_3.1.4 R

run Rscript ex2_barrier.R
```

Expected Results:

```
[1] 6000
[1] 2000
[1] 4000
[1] 5000
[1] 3000
[1] "All Done"
```

University of Virginia | Research Computing

BROADCASTING

Rank 0 sending data to all ranks

Broadcast Concepts

- Suppose . . .
    - One copy of your code generates results that other copies need; or,
    - Each copy of your program needs data from single file. (You do not want the copies to fight over reading the file.)

- Both of these cases can take advantage of the `bcast()` function.
    - `bcast()` allows one rank to "broadcast" data to the other copies.
    - However, you will need to ensure that each copy of your code sets up a variable to receive the data.

MPI Example #3

ex3_broadcast.R:

```
library(pbdMPI, quiet=TRUE)
init()

myRank <- comm.rank()
numProcs <- comm.size()
the_bcaster <- numProcs - 1

if (myRank == the_bcaster){
   values <- matrix(1:24, nrow=4)
} else {
   values <- NULL
}
values <- bcast(values, rank.source =
I                       the_bcaster)

comm.print(values, rank.print=2)
finalize()
```

ex3.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex3_broadcast.R
```

MPI Example #3

ex3_broadcast.R:

```
library(pbdMPI, quiet=TRUE)
init()

myRank <- comm.rank()
numProcs <- comm.size()
the_bcaster <- numProcs - 1

if (myRank == the_bcaster){
   values <- matrix(1:2
} else {
   values <- NULL
}
values <- bcast(values,
I

comm.print(values, rank
finalize()
```

ex3.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00




lf/7.1.0_3.1.4 R

3_broadcast.R
```

Expected Results:

```
COMM.RANK = 2
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

University of Virginia | Research Computing

Hands-on Activity #2

Activities:

1. Modify the line `the_bcaster <- numProcs-1` so that the broadcaster is rank 0.
2. Modify the code so that each rank prints the contents of `values`.
3. Update the code so the copy with rank 0 reads the file myData.txt in the Data directory and distributes it to each of the ranks. Run the program to ensure that each rank received a copy of the data.
   Hint: Use `read.table("./Data/myData.txt")`

SCATTERING

One rank sending subsets of data to all ranks

Scattering Concepts

- Suppose one copy of your program has read the data from a single file, but you want each copy to work on a subset of the data.
- Instead of broadcasting all of the data to every copy, you can send subsets of the to each copy.
- This can be done with the `scatter()` command. However, you will need to ensure that the data is partitioned into a list with the length equal to the number of tasks (i.e., copies of your code).

MPI Example #4

ex4_scatter.R:

```
init()
myRank <- comm.rank()
numProcs <- comm.size()

if (myRank == 0){
   all_values <- read.table("myData.txt")
   col_ndx <- lapply(1:numProcs, function(n)
                seq(n, ncol(all_values), numProcs))
   values_list <- lapply(1:length(col_ndx),
       function(n) all_values[col_ndx[[n]] ])
} else {
   values_list <- NULL
}

values <- scatter(values_list)
comm.print(values, rank.print=2)

### Finish.
finalize()
```

ex4.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex4_scatter.R
```

MPI Example #4

ex4_scatter.R:

```
init()
myRank <- comm.rank()
numProcs <- comm.size()

if (myRank == 0){
   all_values <- read.table("myData.txt")
   col_ndx <- lapply(1:numProcs, function(n)
                   seq(n, ncol(all_values), numProcs))
   values_list <- lapply(1:length(col_ndx),
        function(n) all_values[co
} else {
   values_list <- NULL
}

values <- scatter(values_list)
comm.print(values, rank.print=2)

### Finish.
finalize()
```

Expected Results:

```
    COMM.RANK = 2
     V3 V9 V15 V21
    1  9 33  57  81
    2 10 34  58  82
    3 11 35  59  83
    4 12 36  60  84
```

ex4.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex4_scatter.R
```

GATHER

One rank collecting data from all ranks

Gather Concepts

- There may be times when we want the results from all tasks to be gathered by a single task
- .
- We can use the `gather()` function to accomplish this.
- The default location for gathering the results is the task with rank 0.

MPI Example #5

ex5_gather.R:

```
library(pbdMPI,quiet=TRUE)

init()

myRank <- comm.rank()
x <- sqrt(myRank)
values <- unlist(gather(x))

comm.print(values,all.rank=TRUE)

finalize()
```

ex5.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex5_gather.R
```

MPI Example #5

ex5_gather.R:

```
library(pbdMPI,quiet=TRUE)

init()

myRank <- com
x <- sqrt(myR
values <- unl

comm.print(va

finalize()
```

ex5.slurm:

```
#!/bin/bash

raining

1.0_3.1.4 R

her.R
```

Expected Results:

```
COMM.RANK = 0
[1] 0.000000 1.000000 1.414214 1.732051 2.000000 2.236068
COMM.RANK = 1
NULL
COMM.RANK = 2
NULL
COMM.RANK = 3
NULL
COMM.RANK = 4
NULL
COMM.RANK = 5
NULL
```

Hands-on Activity #3

Activities:

1. Modify the line with the `gather()` function to be as follows:

```
values <- unlist(gather(x, rank.dest=3))
```

   Which rank is now gathering the results?

2. Replace the gather() function with an `allgather()` function as follows:

```
values <- unlist(allgather(x))
```

   What is the difference between the `gather()` and the `allgather()` functions?

POINT-TO-POINT COMMUNICATION

Specific ranks communicate with each other

Point-to-Point Concepts

- There may be times when a task will want to share its result with another, specific task.
- This can be done with the `send()` and `recv()` functions.

MPI Example #6

ex6_point_point.R:

```
library(pbdMPI, quiet=TRUE)
init()

myrank <- comm.rank()
sender <- comm.size() - 1
receiver <- sender - 1

if (myrank == sender) {
 msg <- paste("Hi from:", sender)
 send(msg, rank.dest=receiver)
 } else if (myrank == receiver) {
 buf <- recv(rank.source=sender)
 }
 comm.print(paste("I am:",receiver),
                   rank.print=receiver)
 comm.print(buf, rank.print=receiver)

finalize()
```

ex6.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00



module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex6_point_point.R
```

MPI Example #6

ex6_point_point.R:

```
library(pbdMPI, quiet=TRUE)
init()

myrank <- comm.rank()
sender <- comm.size() - 1
receiver <- sender - 1

if (myrank == sender) {
 msg <- paste("Hi from:", sender)
 send(msg, rank.dest=receiver)
 } else if (myrank == receiver) {
 buf <- recv(rank.source=sender)
 }
 comm.print(paste("I am:",re
                rank.print=r
 comm.print(buf, rank.print=

finalize()
```

ex6.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00



module purge
module load goolf/7.1.0_3.1.4 R

                    pt ex6_point_point.R
```

```
[1] "I am: 4"
COMM.RANK = 4
[1] "Hi from: 5"
```

Hands-on Activity #4

Background Information:

Suppose we have a list of integers from 0 to 11, and we want each value to be updated according to the following rules.

Activities:

1. Write a script where each task starts with the value equal to its rank. Have the ranks with odd values send a copy of its value to the rank one less than itself. The tasks with an even rank receives the value and averages it with its own value. So only the even numbered task will update their values. Gather the results and print the new values.
2. Modify the script so that even ranks will pass their values to the odd ranks. Have the ranks 1 – 10 average their value with the two neighbor values. Have the ranks 0 and 11 average their value with the one neighbor value. Gather the results and print the new values.

MAP-REDUCE COMMUNICATION

Combine results in a pre-described manner

All Reduce Concepts

- There may be times when we want to combine the values from the different task using a basic operation, like summation.
- We can use the reduce or allreduce functions to accomplish this.
- The defaullt operation is sum. Other operations include
  - prod
  - max
  - min
  - land (logical and)
  - lor (logical or)

MPI Example #7

ex7_allreduce.R:

```r
library(pbdMPI,quiet=TRUE)
init()

myRank <- comm.rank()

# Define x
x <- 1

# Default reduce will sum the
# values
y <- allreduce(x)
Comm.print(y)

finalize()
```

ex7.slurm:

```bash
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

srun Rscript ex7_allreduce.R
```

MPI Example #7

ex7_allreduce.R:

```
library(pbdMPI,quiet=TRUE)
init()

myRank <- comm.rank()

# Define x
x <- 1

# Default reduce will sum the
# values
y <- allreduce(x)
Comm.print(y)

finalize()
```

ex7.slurm:

```
#!/bin/bash

#SBATCH -A rivanna-training
#SBATCH -p parallel
#SBATCH -N 2
#SBATCH -n 6
#SBATCH -t 00:03:00


module purge
module load goolf/7.1.0_3.1.4 R

           pt ex7_allreduce.R
```

```
COMM.RANK = 0
[1] 6
```

Hands-on Activity #5

Activities:

1. Modify the code so that x is assigned `myRank + 1` and the allreduce() line includes the product operation. For example:

```
y <- allreduce(x, op='prod')
```

   Run the code and observe that a multiplication has been performed.

2. Modify the code so that x is assigned `2*myRank` and the allreduce line includes the maximum operation. For example:

```
y <- allreduce(x, op='max')
```

   Run the code and observe that the maximum value has been printed.

3. Modify the code so that x is assigned the sequence `1:5` and the allreduce() line includes the summation operation. For example:

```
y <- allreduce(x, op='sum')
```

   Run the code and observe the results. Are they what you expected?

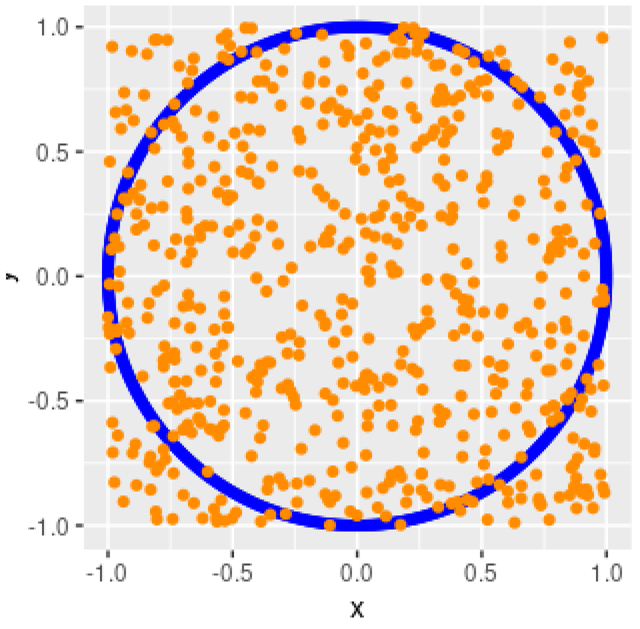
Final Activity (Part 1)

Background Information:

A fun way to estimate the value of pi is to think about throwing darts at a unit circle enclosed in a square.

Some darts would land inside the unit circle, but all darts would be inside the 2x2 square.

If we consider a "large" number of random darts, then the ratio of those landing inside the circle versus the total number darts is similar to the ratio of the area of the circle versus the area of the square.



Mathematically, this ratio is $\frac{\pi}{4}$.

In other words: $\frac{\#points\ \ in\, Circle}{Total\ \#points} \approx \frac{\pi}{4}$ or $\pi \approx 4 * \frac{\#points\ \ in\, Circle}{Total\ \#points}$

Final Activity (Part 2)

Activity:

The code to estimate pi is very simple; however, we would need an extremely large number of random (x,y) pairs to get a decent estimate for pi.

How could we convert this code to take advantage of parallelism using MPI?

```
### Randomly choose N pairs of
### (x,y) coordinates in 2x2 square
N <- 1000
x <- runif(N, -1.0, 1.0)
y <- runif(N, -1.0, 1.0)

### Determine how many are within
### the unit circle
distance <- sqrt(x*x + y*y)
numInUnitCircle <-
length(which(distance <= 1.0))

### Estimate pi
myPi <- 4*numInUnitCircle / N

### Print result
print(myPi)
```

NEED HELP?

Research Computing Zoom Office Hours

https://www.rc.virginia.edu/support/#office-hours

Tuesdays:	3 pm – 5 pm
Thursdays:	10 am – noon

Or, contact us through the forms at:

https://www.rc.virginia.edu/support/