Using Open MPI with R

Peng Zeng*
Department of Mathematics and Statistics, Auburn University

November 10, 2018

Abstract

Open MPI is popularly used for parallel computing nowadays. This tutorial discusses some simple examples to demonstrate how to use open MPI with R on Linux or Mac OS. It covers how to install open MPI, some basics on MPI programming in C, how to build a shared library to use in R, and how to run a MPI program.

1 Introduction

Message passing interface (MPI) is a standard designed for parallel computing on a wide variety of architectures. It is the dominant model used in high-performance computing nowadays. Open MPI (`https://www.open-mpi.org`) is an open source implementation for MPI. Currently, open MPI only supports Linux and Mac OS. The support for Windows has been discontinued.

Do not confuse open MPI with openMP. The latter, standing for open multi-processing, is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming. It is a feature that has been implemented in many compliers, including Visual C++, GCC, etc. On the contrary, parallel processes in MPI usually do not share memory and exchange data by passing messages between them.

2 Installation of Open MPI

Open MPI can be installed following the steps below on Linux or Mac OS. Note that open MPI has stopped supporting Windows.

1. Download the latest stable release of open MPI on `https://www.open-mpi.org`. I downloaded `openmpi-3.1.3.tar.gz`.

2. Uncompress the file using the following command. It will create a directory `openmpi-3.1.3`.

```
tar -xzf openmpi-3.1.3.tar.gz
```

3. Enter the directory.

*Peng Zeng, Department of Mathematics and Statistics, Auburn University, Auburn, AL 36830, USA. Email: `zengpen@auburn.edu`.

```
cd openmpi-3.1.3
```

4. Configure the installation settings using the following command. You may change the installation directory to a different one. If you choose to install it in your home directory, the command `sudo` is not needed.

```
sudo ./configure --prefix="/usr/local/openmpi"
```

5. Compile the software using the following command.

```
sudo make
```

6. Install open MPI using the following command.

```
sudo make install
```

7. Update environment variables `$PATH` and `$LD_LIBRARY_PATH`. I added the following two lines in the end of `~/.bash_profile` (Mac OS) or `~/.bashrc` (Ubuntu).

```
export PATH="$PATH:/usr/local/openmpi/bin"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/openmpi/lib"
```

In order to check whether open MPI has been installed correctly, let us compile and run `hello.c`. The detail of this function will be explained in the next section. Type the following to compile.

```
mpicc -o hello hello.c
```

It will create a program named `hello` in the current directory. Type the following line to run the problem.

```
mpirun -np 2 ./hello
```

If everything works fine, the following lines will be displayed on screen.

```
Hello world from processor ProcessName, rank 0 out of 2 processors
Hello world from processor ProcessName, rank 1 out of 2 processors
```

Notice that the number 2 in the command specifies the number of processes. If your computer has more than 2 cores, you may change 2 to other numbers to see what happens. Notice that if you specify a number larger than the number of cores on your computer, an error message will be displayed.

3 First Example in C

The following is the content of `hello.c`.

```
/***********************************************************************
 * hello.c
 * example from
 * http://mpitutorial.com/tutorials/mpi-hello-world/
 ***********************************************************************/

#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    /* Initialize the MPI environment */
    MPI_Init(NULL, NULL);

    /* Get the number of processes */
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    /* Get the rank of the process */
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    /* Get the name of the processor */
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    /* Print off a hello world message */
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    /* Finalize the MPI environment. */
    MPI_Finalize();
}
```

Some explanations of `hello.c` are as follows.

- In order to use MPI, one needs to include all MPI-related function calls within `MPI_Init()` and `MPI_Finalize()`.

```
    MPI_Init(NULL, NULL);
    /* write your codes here */
    MPI_Finalize();
```

- Get the rank (or ID) of the process using `MPI_Comm_rank()`, and get the total number of processes using `MPI_Comm_size()`.
- In the SPMD (single program, multiple data) scenario, all processes run the same codes, but have different data (or memory). In order to let different processes do different jobs, write the code as follows.

```
if(rank == 0)
{
    /* process 0 does these jobs */
}
else if(rank == 1)
{
    /* process 1 does these jobs */
}
else if(rank == 2)
{
    /* process 2 does these jobs */
}
else
{
    /* remaining processes do these jobs */
}
```

Compile the code using `mpicc`, which is essentially a wrapper of other compilers such as `gcc` (usually in Linux) or `clang` (usually in Mac OS). You can check the compile options for `gcc` using

```
mpicc -showme:compile
```

The linking options can be obtained using

```
mpicc -showme:link
```

You can get all options using

```
mpicc --showme
```

In order to run the program in MPI, need to type

```
mpirun -np 2 your_program arguments_to_your_program
```

The option `-np` specifies the number of processors, which should be less than or equal to the number of cores on your computer.

4 Open MPI with R

This section discusses an example to calculate the sum of a vector using open MPI. Suppose that a vector x has n components $x_0, x_1, \ldots, x_{n-1}$. Assume that the number of processes is k. Let

$a = n/k$ be an integer. If a is not an integer, we need to modify the algorithm accordingly. For simplicity, we just assume a is an integer.

The idea is to let each process calculate the partial sum of a numbers. The process 0 calculates the sum of $x_0, \ldots, x_{a-1}$; the process 1 calculates the sum of $x_a, \ldots, x_{2a-1}$; $\ldots$; the process i calculates the sum of $x_{ia}, \ldots, x_{(i+1)a-1}$; $\ldots$. Finally, the partial sums calculated by different processes are summed together to yield the sum of all numbers.

4.1 C function

The function is executed independently for different processes as follows. Each process will get a copy of `n` and `x` in the beginning of the function call. Notice that `n` and `x` for different processes are saved in different memory locations. During the function execution, different processes calculate the sum of different elements of the vector, which results in different values of `xsumi`. Finally, the function `MPI_Allreduce()` sums `xsumi` together and save the result in `xsum`. Thus, all function calls from the different processes return the same value.

```
#include <R.h>
#include <mpi.h>

void sum_mpi(const int *n, const double *x, double *xsum)
{
    MPI_Init(NULL, NULL);
    /* Get the number of processes and the rank of the process */
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    double xsumi = 0.0;
    int block_size = (*n) / world_size;
    int pos = world_rank * block_size;
    for(int i = 0; i < block_size; i++, pos++)
        xsumi += x[pos];

    MPI_Allreduce(&xsumi, xsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Notice that the above code has a potential bug when the length of the vector n is not a multiple of the number of processes k. But the code is enough for the demonstration of using open MPI with R.

The above function demonstrates a typical mechanism for parallel computing. Different processes run the same function on different portions of data, and the results from different processes are aggregated together to produce the final answer. Some comments are as follows.

- It is common to use the first process (rank = 0) to handle the input or output. For example, in the beginning of calculation, you may add the following lines.

```
    if(world_rank == 0)
    {
        printf("Start calculation.\n");
    }
```

In the end of calculation, you may add the following lines.

```
if(world_rank == 0)
{
    printf("Stop calculation.\n");
}
```

- If the dataset is large, it is common to partition and save the data in several files. Each process will load one file and work on that portion of data.

- If it is necessary to synchronize all processes, we may add the following line in a proper place in C function. The process reaches this line early will pause and wait for other processes. The process will resume execution until all processes reaches this line.

  ```
  MPI_Barrier(MPI_COMM_WORLD);
  ```

  The function `MPI_Allreduce()` has this mechanism built-in, because it will wait to receive data from all processes and then aggregate them together.

- It is possible to send data from one specific process to another specific process. For more detailed tutorials on MPI, refer to `http://mpitutorial.com/`.

4.2 Compile C functions: First Approach

It is tricky to compile the C code and create a shared library for R to use. There are two different approaches we can use. This subsection discusses `R CMD SHLIB`, while the next subsection discusses `mpicc`.

The standard way of create a shared library for R is to use `R CMD SHLIB`. We need to specify the correct directory for include files and linking library for compiling and linking open MPI.

Run the following commands to define two environment variables, where `$MPI_CPPFLAGS` contains the options for compiling and `$MPI_LDFLAGS` contains the options for linking.

```
export MPI_CPPFLAGS=$(mpicc -showme:compile)
export MPI_LDFLAGS=$(mpicc -showme:link)
```

Create a file named `Makevars` with the following two lines.

```
PKG_CFLAGS = $(MPI_CPPFLAGS)
PKG_LIBS = $(MPI_LDFLAGS)
```

Type the following command to compile the C code and create a shared library named `sum_mpi.so`.

```
R CMD SHLIB sum_mpi.c
```

4.3 Compile C functions: Second Approach

This subsection discusses how to create a shared library using `mpicc`. We need to get the options for R if some R functions are used. The example `sum_mpi.c` does not use any features in R. But it is common that a C function calls some APIs from R, or maybe use functions in BLAS or LAPACK via R.

We need correct options to compile and link R, which can be obtained by the following commands.

```
R CMD config --cppflags
R CMD config --ldflags
```

You can type the following to get a list of available environment variables defined in R.

```
R CMD config
```

For example, the following two lines yield the correction options to link BLAS and LAPACK libraries via R.

```
R CMD config BLAS_LIBS
R CMD config LAPACK_LIBS
```

It is convenient to get all necessary R environment variables and save them in a file, which can be included in a makefile. The following script file will get most necessary R environment variables and save them in a file named `R_env_variables`.

```
./R_config
```

Prepare a makefile as follows.

```
-include R_env_variables

EXECS = sum_mpi
MPICC = mpicc

all: $(EXECS)

sum_mpi: sum_mpi.o
	$(MPICC) -shared -o sum_mpi.so sum_mpi.o $(R_LINK)

sum_mpi.o: sum_mpi.c
	$(MPICC) $(R_COMPILE) -c sum_mpi.c -o sum_mpi.o
```

Then type the following to compile and generate a shared library.

```
make
```

If everything works fine, it creates a shared library named `sum_mpi.so` in the current directory.

4.4 R function

It does not matter whether you use `R CMD SHLIB` or `mpicc`, as long as you successfully create a shared library named `sum_mpi.so`. We load the shared library in R using function `dyn.load()`.

The following codes are the content of `sum_mpi.R`. It defines a function `sum_MPI`, which calls the C function defined in `sum_mpi.c`. For demonstration, we simply calculate the sum of numbers from 1, 2, . . . , 300.

```
dyn.load(paste0("sum_mpi", .Platform$dynlib.ext));

sum_MPI = function(x)
{
    .C("sum_mpi", as.integer(length(x)), as.double(x),
        xsum = double(1))$xsum;
}

y = 1:300
ans0 = sum(y);          # calculated using R built-in function
ans1 = sum_MPI(y);      # calculated using parallel computing

cat("The sum is", ans1, "and the actual sum is", ans0, "\n");
```

Run the R code using the following line.

```
    mpirun -np 2 R --slave -f sum_mpi.R
```

Equivalently, we can also use

```
    mpirun -np 2 Rscript --slave sum_mpi.R
```

If everything works fine, the screen will display the following.

```
The sum is 45150 and the actual sum is 45150
The sum is 45150 and the actual sum is 45150
```

Note that although the R function `cat()` is beyond `MPI_Init()` and `MPI_Finalize()`, which are contained in C function `sum_mpi()`, it is still executed by both processes. The reason is that `MPI_Init()` and `MPI_Finalize()` only marks the beginning and ending positions where we can call MPI related functions. There are essentially two independent instances of R are running, one by each process, as specified by `mpirun`.

5 R Package Rmpi

The package `Rmpi` provides a R interface to MPI. You can try to install the package directly within R using the following function.

```
    install.packages("Rmpi")
```

If the installation does not succeed with an error message such as mpi.h or mpi library could not be found, you need to install it manually. Firstly download the source code from CRAN. I downloaded `Rmpi_0.6-8.tar.gz`. Recall that I installed open MPI in the directory of `/usr/local/openmpi`. Install `Rmpi` using the following command.

```
R CMD INSTALL Rmpi_0.6-8.tar.gz --configure-args=--with-mpi=/usr/local/openmpi
```

When using `Rmpi`, the main process will spawn some processes as workers. Hence for a total of k processes, one is the main process (master process) and the remaining $k-1$ are workers (slave processes). The calculation is usually conducted by all workers, while the main process only controls input, output, dispatch jobs, and collect results.

I do not explore how to use `Rmpi` right now.

6 Conclusion

Open MPI is not formidable. If you can invest some time to figure it out, it will boost your programming skills to a next level, and definitely open a new world to you.