6. Parallel programming with MPI

[1]

[1] M. Ganesh, MATH440/540, SPRING 2018

[2] MPI

- **Almost all parallel scientific computing codes use the Message Passing Interface (MPI) standard for execution of various tasks on HPC multi-core cluster systems**
- **There are several documents available for MPI (many free online). For details of the MPI standard, documents, books, some codes, see** `http://www.mpi-forum.org/` **and** `http://www.mcs.anl.gov/mpi/`
- **There are several implementations of the MPI standard:**
  - ⋆ **OPEN-MPI (MIO, BlueM, and Sayers Lab), see** `http://www.open-mpi.org/`
  - ⋆ **MPICH2 (MIO and BlueM), see** `http://www.mcs.anl.gov/research/projects/mpich2/`
  - ⋆ **MVAPICH (available on MIO and BlueM), see** `http://mvapich.cse.ohio-state.edu/`
- **The MPI version on Sayers Lab and on MIO/AuN Linux machines is 3.1**

[2] M. Ganesh, MATH440/540, SPRING 2018

- [3]**MPI is a special class of communications library that is independent of the HPC systems and can be considered similar to other communications library such as TCP/IP**

  item MPI 2.0 standard (1998) is extensions of MPI 1.* (1994-2008)
- **Version 2.2 of the MPI standard was approved 2009**
- **Version 3.0 (2012) is an extension of MPI-2.2**
- **MPI is a library of routines that can be called from Fortran, C, and C++ codes. (Bindings are used to call MPI routines from MATLAB, PYTHON, JAVA etc.)**
- **In a message passing program, each processing core runs a sub-program (written in sequential languages such as C or Fortran)**
- **Typically, MPI is used in SPMD (Single Program, Multiple Data) mode. That is, all processing cores run a single executable file**
- **Although SPMD is not required by MPI (in theory MPMD is allowed), most implementations of MPI support only SPMD**
- **The sub-programs from the executable have different locations (distributed memory), different data and variables are private**

[3]M. Ganesh, MATH440/540, SPRING 2018

- [4]Messages are packets of data moving between sub-programs/cores
- Information list required for the message passing system includes:
  - ⋆ Sending processing cores and receiving processing cores
  - ⋆ Source datatype and destination datatype
  - ⋆ Source buffer size and destination buffer size
- Sub-programs must be linked with an MPI library and the total program must be started with the MPI startup tool
- Messages need to have addresses to be sent to
- Addresses in MPI are ranks for the MPI processes
- All messages must be received
- A simple form of message passing is: one processing core sends a message to another (pairwise). Such communications are called `point-to-point`
- Communications involving several processing cores are called `collective`. Examples include master core broadcasting variables to several cores and receiving information from several cores for reduction operations

[4]M. Ganesh, MATH440/540, SPRING 2018

- [5]**Communications (point-to-point or collective) need not be synchronized, leading to synchronous and asynchronous (buffered) communications**
- **In synchronous communication, for example, the sender gets an information that the message is received**
- **In asynchronous communication, for example, the sender knows only that the message has left**
- **In MPI there are commands available to ask specifically for one of these communications**
- **If you make error in some MPI based codes, say, in send/receive calls, a communication may lead your program to hang**
- **However, there are many MPI operations that can never hang but will lead to return back on encountering errors with some form of error messages**
- **The above two points depend on the MPI implementation (such as OPEN-MPI, MPICH2,.....)**
- **It is important to be aware of the above for programming with MPI**

[5]M. Ganesh, MATH440/540, SPRING 2018

- [6]**All communications occur within a communicator**

- **MPI communicators can have any number of processing cores**

- **A typical/default MPI communicator used in MPI programming is:** `MPI_COMM_WORLD`

- **All MPI command names start with** `MPI_` **or** `PMPI_` **(for profiling) and hence avoid variables in your code starting with these two strings**
- **MPI interfaces (commands) are different for C, C++, and Fortran**
- **Use only one interface and do not mix these. (Although C interface can be used from C++, avoid mixing.)**
- **The MPI calling commands for C, C++, and Fortran look similar but not same, as we shall see below**
- **The similarity facilitates easier learning and using of interface in other languages from a known language MPI interface**

[6]M. Ganesh, MATH440/540, SPRING 2018

- [7]**In Fortran interface, MPI procedures are subroutines and hence require** `CALL` **from programming units:** `CALL MPI_******(..........)` **(Only a few exceptions, for example,** `my_start_time = MPI_Wtime()` **.)**
- **In C interface, MPI procedures are called as** `MPI_******(......);`
- **In C++ interface, MPI procedures are called as** `MPI::*******(.......);`
- **Almost all** `MPI_*` **procedures (in C, C++, and Fortran interface) return an error code that is an integer.**
  If successful, the return value of this integer is zero
- **Below, we use an integer variable** `ierror` **to get this error and we assume that this variable is declared as an integer in all calling programming units**
- **In C and C++, respectively, we *may* prefer to get this error code as** `ierror = MPI_******(..........)` **and** `ierror =MPI::*******(..........)`
- **However, in Fortran interface it is important to include the variable** `ierror` **in MPI call statements. This variable is the last in the list of MPI subroutine call arguments:**
  `CALL MPI_******(...........,ierror)`

[7]M. Ganesh, MATH440/540, SPRING 2018

- [8]**In C, C++, and Fortran programming with MPI:**
  - **There must be a statement to include the MPI header files (`INCLUDE` in C, C++, and Fortran or `USE` in F90+ with MPI2.*)**
  - **The MPI should be initialized using the procedure `MPI_INIT` before calling any other MPI procedures**
- **Basic MPI interface in Fortran:**
  - `USE mpi  !For MPI 1.* version, use INCLUDE ‘‘mpif.h’’`
  - `CALL MPI_Init(ierror)`
- **Basic MPI interface in C:**
  - `#include ‘‘mpi.h’’`
  - `MPI_Init(&argc, &argv);`
- **Above and below assume a program unit of the form**
  `int main (int argc, char *argv[]) {`
- **Basic MPI interface in C++:**
  - `#include ‘‘mpi.h’’`
  - `MPI::Init(argc, argv);`

[8]M. Ganesh, MATH440/540, SPRING 2018

- [9]**Below is a simple programming example to initialize the MPI communicator and ask each processing core to print "hello, you called me" and close the communicator with** `MPI_FINALIZE`
- **Fortran, C, and C++ programming with MPI using basic MPI calls:**
- **F90+ programming with MPI (**`hello_calls.f90`**):**

```
PROGRAM hello_calls
USE mpi
IMPLICIT NONE
INTEGER ierror
CALL MPI_Init(ierror)
WRITE(*,*) 'Hello, you called  me'
CALL MPI_Finalize(ierror)
END PROGRAM hello_calls
```

- Exercise 1: **Generalize the above F90+ code (C/C++ programmers, in addition, below C/C++ codes) to print *once* the MPI version and sub-version (i.e., 2.1 or ...) of the parallel computing environment on which the above (and below) code is executed**
(HINT: Use intrinsic MPI variables `MPI_VERSION, MPI_SUBVERSION`**)**

[9]M. Ganesh, MATH440/540, SPRING 2018

- [10]**C programming with MPI (`hello_calls.c`):**

```
#include <stdio.h>
#include  "mpi.h"
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   printf ("Hello, you called me\n");
   MPI_Finalize();
   return 0;
}
```

- **C++ programming with MPI (`hello_calls.cpp`):**

```
using namespace std;
#include <iostream>
#include "mpi.h"
int main (int argc , char * argv [] ){
     MPI::Init ( argc , argv ) ;
     cout <<  "Hello, you called me" << endl;
     MPI::Finalize () ;
     return 0 ;
}
```

[10]M. Ganesh, MATH440/540, SPRING 2018

- **[11]To compile codes with MPI commands, use:**
  - ⋆ `mpif90` **(MIO, BLUEM and Sayers Lab) for FORTRAN 90+ codes**
  - ⋆ `mpicc` **for C and** `mpiCC` **for C++ codes (MIO, BLUEM and Sayers Lab)**
- **In particular, for simple codes such as** `hello_calls.*` **use, say,**

```
mpif90 -o hello_calls_exe hello_calls.f90

mpicc -o hello_calls_exe hello_calls.c

mpiCC -o hello_calls_exe hello_calls.cpp
```

- **For a project with several procedures, use an appropriately modified version of the** `makefile` **discussed in Chapter 4**

[11]M. Ganesh, MATH440/540, SPRING 2018

- [12]**To run a project executable file say,** `hello_calls_exe`**, on one Sayers Lab machine, with** `xx` **cores (max. value of** `xx = 8`**) with default MPI implementation, OPEN-MPI, use the command**

  ⋆ `mpiexec -np xx ./hello_calls_exe`

- **To run (using** `yy` **cores) on specific Sayers Lab machines, say,** `ch215l-01`**,** `ch215l-02`**,** `ch215l-03` **(starting from, say,** `ch215l-01` **you are submitting a job) create a file, say,** `lab_ma_file` **with these machine names and replace the above command (on** `ch215l-01`**) with**
  `mpiexec -machinefile lab_ma_file -np yy ./hello_calls_exe`

- **Before using the above command, it is important to make sure (at least once) that you can ssh to the required machines (**`ch215l-02`**,** `ch215l-03`**) from the machine you are submitting the job (**`ch215l-01`**)**

- **If the machine file** `xxx_machine` **contains only machine names, all available cores in the machine names will be counted for running the job**

- **To use only** `aa` **cores on** `ch215l-01`**,** `bb` **cores on** `ch215l-02` **and** `cc` **cores on** `ch215l-03`**, replace** `ch215l-01` **in** `lab_ma_file` **with** `ch215l-01 cpu=aa` **and similarly for other machines**

[12]M. Ganesh, MATH440/540, SPRING 2018

- [13]**To run a project executable file say, `project_exe`, on `nn` MIO/BlueM-aun compute nodes, with `xx` cores using default MPI implementation, OPEN-MPI, you need to submit a job to the MIO/BlueM-aun (SLURM) queue system from the MIO/BlueM-aun head node (the node you get when you login to MIO/BlueM-aun). [Max. value of: `xx = 12`, to use *any* MIO node, including our six old nodes (pre-2015) ; `xx = 24`, for each of our new (2015) MIO nodes; `xx = 16`, for each aun node.]**

- **To this end, you need to first create a script file, say, `my_project.slurm` with the details given below (see Page 14 &15) and submit the job using the command (on MIO/BlueM-aun head node)**
  `sbatch my_project.slurm`.
  In order to use only our old (pre-2015) MIO nodes, use the command
  `sbatch -p mganesh my_project.slurm`
  In order to use only our new (2015) MIO nodes, use the command
  `sbatch -p mganesh2015 my_project.slurm`
- **Below, we assume that we would like our job in the queue system to be shown as `my_project`; the screen output to be recorded in a file `my_project.out` ; any error in running the job to be recorded in a file `my_project.err`**

[13]M. Ganesh, MATH440/540, SPRING 2018

- **Below, replace `nn` and `xx` with appropriate numbers. For example, to use a total of 24 cores to run `project_exe` with two MIO nodes. using all 12 cores within each node, replace `nn` with 2 and `xx` with 12**
- **Below we assume that the maximum walltime required to run job is `hh` hours, `mm` minutes and `ss` seconds. For example to require only a maximum of 20 minutes to run `project_exe`, replace `hh` with 00, `mm` with 20 and `ss` with 0**

```
#!/bin/bash -x
#SBATCH --job-name="my_project"
#comment      = "Hello ..."
#SBATCH --nodes=nn
#SBATCH --ntasks-per-node=xx
#Multiply nn and xx (denoted below as nn*xx) and use that  in the line
#SBATCH --ntasks=nn*xx
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=hh:mm:ss
#SBATCH -o project.out
#SBATCH -e project.err
```

```
# Go to the directoy from which our job was launched
cd $SLURM_SUBMIT_DIR

# Create a short JOBID base on the one provided by the scheduler
JOBID=‘echo $SLURM_JOBID’

# Save a copy of our environment and script
cat $0 > script.$JOBID
printenv  > env.$JOBID

# Run the job.
# The echo will go into the standard output for this job
# The standard output file will end up in the directory
# from which the job was launched.
echo "running job"
srun project_exe
echo "job has finished"
```

Remark 7.1 • ***For submission of your job on BlueM (Aun or Mc2) nodes (using our class account), add the following line***

```
#SBATCH --account=math540s18
```

- **To check the status of your jobs, type (on MIO/BlueM-aun head node)**
  `squeue -ulogin_name`
- **To see all jobs, remove `-ulogin_name` in the above command**
- **The above commands will give a jobid number for each of your job of the form *******
- **To delete your job (with jobid *****) from the queue, type**
  `scancel *****`
- **You may also check your job (and other MIO/BlueM status) at tuyo.mines.edu/ganglia/ mindy.mines.edu/ganglia/**

[14] More MPI calls for parallel scientific computing

- `MPI_COMM_WORLD` **is a predefined handle in** `mpi.h`, `mpif.h` **and in F90+** `mpi` **module**
- **Below, unless specified, assume this to be the choice communicator**
- **Below, in using all of the following MPI commands, we assume that all variables (with non-CAPS letters) are appropriately declared in all calling program units**
- **An important component in parallel codes is to identify each processing core, known as rank, to allocate, compute, send/receive messages, execute sub-programs etc. (If a total of P processing cores are allocated for a job, then the cores are ranked as $0, 1, 2, \cdots, P-1$.) The MPI procedure** `MPI_Comm_rank` **is useful to get the rank of each core**
- **Find the rank of processing cores - Fortran, C, C++ with MPI:**
  - `CALL MPI_Comm_rank (MPI_COMM_WORLD,my_rank,ierror)`
  - `MPI_Comm_rank (MPI_COMM_WORLD,&my_rank)`
  - `my_rank = MPI::COMM_WORLD.Get_rank();`

[14] M. Ganesh, MATH440/540, SPRING 2018

- [15]**Henceforth, we use the variable `master` to refer to the first core in the communicator**
- **For this purpose, create a variable `master` in your mpi programs, and this variable should correspond to, based on the above commands, `my_rank = 0`**
- **Another important component in MPI is to identify the total number (size) of the processing cores contained within a communicator**
- **Total number of processing cores requested (when submitting an executable file of a project) determines the size of a communicator**

- **The communicator and its size is induced by the command `mpirun` or `mpiexec` that executes the project**
- **Find the size of `MPI_COMM_WORLD` - Fortran, C, C++ with MPI:**
  - `CALL MPI_Comm_size (MPI_COMM_WORLD,num_cores,ierror)`
  - `MPI_Comm_size (MPI_COMM_WORLD,&num_cores);`
  - `num_cores = MPI::COMM_WORLD.Get_size() ;`

[15]M. Ganesh, MATH440/540, SPRING 2018

- [16]Exercise 2: **Generalize your code(s) in Exercise 1 so that *each* processing core should print *instead* the following message:**

  ⋆ `'Hello, I am processing core < ... > in a communicator consisting total of < ... > cores running within the node_name < ... >'`

  by filling appropriate data in < ... >, obtained from executing the code(s) on parallel computing environments. for example on MIO and various on Sayers Lab machines

  ⋆ **and the master core should print an additional message:**
  `'Hello, thanks for assigning me as the master within the present communicator.'`

(Hint: Use `MPI_Get_processor_name(procname,namelen,ierror)` **to get the node name and the intrinsic MPI variable** `MPI_MAX_PROCESSOR_NAME` **to declare the maximum length of the character** `procname`**.)**

[16]M. Ganesh, MATH440/540, SPRING 2018

- [17]Exercise 3(Local/Partial sum): **(Using only 4 `MPI_***` commands)**
  Let `a` be a vector with $N \geq 100,000$ components.
  Suppose that, for example, $a(i) = i^2, i = 1, \cdots, N$.
  Assume that we have access to $P \geq 2$ processing cores.
  Our first task is to let each of the first $P-1$ processing cores to allocate a vector of dimension N/P and the last P-th core to allocate a vector of dimension $N/P + r$, where $r = 0$ if N is divisible by P and $r > 0$ is the reminder term in the division N/P.
  The second task is to let: the first core (that is, `master`) create the first N/P components of the vector `a`; the second core create the next N/P components of `a` and so on.
  (Thus, the last core should create the last $N/P + r$ components of `a`.)
  Our third task is to let *each* processing core to sum up all the created sub-components of `a` and store in the variable `local_sum`. (Thus there are P private `local_sum`.) Our final task includes: printing each rank and the associated `local_sum`. (You may also consider printing any error in computing `local_sum` using the sum of squares formula.)
  Write a parallel program with MPI to solve the above local summation problem. Test your code with $N = 100,000$ and $P = 4, 8, 16, 32, 64$

[17]M. Ganesh, MATH440/540, SPRING 2018

- [18]**The next MPI task, for example, in the previous exercise is for all `non-master` cores to send their private variable `local_sum` and the `master` to appropriately receive the local/partial sum values and create a variable `full_sum` with actual value $\sum_{i=1}^{N} a(i)$**
- **The `MPI_SEND` and `MPI_RECV` commands facilitate the above task (to send/receive data from any core to any other core)**
- **For this purpose, it is important that sending/receiving processing cores are aware of the size and type of the data**
- **MPI transfers use a vector, 1-D array, called henceforth `buffer`**
- **Each `buffer` variable is a contiguous array**
- **The length of `buffer` henceforth is called `count`. (In the last exercise, `count` of `buffer = local_sum` is 1 and that of `buffer = a` is $N = 100,000$.)**
- **The base element of `buffer` (that is each component of the vector) is a scalar with a datatype and it is the datatype of `buffer`(In Exercise 2 and 3, the datatype of N is integer, `procname` is character and we assume that of `local_sum` and `a` to be of type `REAL(KIND = KIND(0.0D0))`**
- **Datatype is passed as a separate argument in MPI transfers**

[18]M. Ganesh, MATH440/540, SPRING 2018

- [19]**In MPI, datatypes are constants and not language type, but datatypes** `must match` **. There is a large set of built-in MPI data types**
- **Some basic MPI datatypes for Fortran programming with MPI are:** `MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX. MPI_DOUBLE_COMPLEX, MPI_CHARACTER, MPI_LOGICAL`
- **In F90+ variables, it is better to avoid using** `DOUBLE` **and instead use** `KIND` **numbers. Above,** `DOUBLE` **corresponds to** `KIND = KIND(0.0D0)`
- **MPI 2.* supports the type** `REAL(KIND=SELECTED_REAL_KIND(xx,yy))` **and we shall discuss this (and other derived data types) later**
- **If you use old type, say** `REAL*8`**, in your Fortran codes (avoid), you may use** `MPI_REAL8` **(avoid). (In general** `type*x` $--> $ `MPI_typex`**.)**
- **Some basic MPI datatypes for C/C++ programming with MPI are:** `MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_INT, MPI_UNSIGNED, MPI_LONG, MPI_UNSIGNED_LONG, MPI_UNSIGNED_CHAR, MPI_CHAR`
- **MPI 2.* supports** `MPI_SIGNED_CHAR`**. (For C++, replace** `MPI_` **with** `MPI::`**)**
- **Some of the additional recommended datatypes in C++ are** `MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, MPI::LONG_DOUBLE_COMPLEX`

[19]M. Ganesh, MATH440/540, SPRING 2018

- [20]**MPI send messages come in envelopes**
- **MPI send envelops contain:**
  the source core (set automatically by the call `MPI_SEND`),
  the destination processing core (henceforth, we use the variable `to`),
  the communicator and an identifying tag. (The tag is an arbitrary integer of your choice; henceforth we use variable `my_tag` .)
- **MPI receive action includes:**
  the source processing core (henceforth, we use the variable `from`),
  the identifying tag given by the variable `my_tag`
  and a status information (henceforth we use the variable `my_status`)
- **For Fortran programming with MPI, the variable**
  `my_status` is an integer array and get this by declaring:
  ```
  INTEGER, DIMENSION(MPI_STATUS_SIZE) ::  my_status
  ```
- **For C/C++ programming with MPI, the status is a structure and to get `my_status`, respectively use:**
  ```
  MPI_Status my_status
  MPI::Status my_status
  ```

[20]M. Ganesh, MATH440/540, SPRING 2018

- **MPI commands for a processing core to send**
  - ⋆ **an array (say,** `buffer`**), with total size (say,** `count`**)**
  - ⋆ **and datatype (say,** `MPI_DOUBLE_PRECISION or MPI_DOUBLE`**)**
  - ⋆ **to another processing core (say,** `to`**), with a tag (say,** `my_tag`**)**
  - ⋆ **using a communicator (say,** `MPI_COMM_WORLD`**)**

  in Fortran, C, and C++ programming with MPI are respectively:

```
CALL MPI_Send(buffer, count, MPI_DOUBLE_PRECISION, to, my_tag,
                    MPI_COMM_WORLD, ierror)
MPI_Send(buffer, count, MPI_DOUBLE, to, my_tag, MPI_COMM_WORLD);
MPI::COMM_WORLD.Send(buffer, count, MPI_DOUBLE, to, my_tag);
```

- **MPI commands to receive** `buffer` **(with above properties) sent by a processing core (say,** `from`**) by a processing core with status (say,** `my_status`**) in Fortran, C, and C++ are respectively given by:**

```
CALL MPI_Recv(buffer, count, MPI_DOUBLE_PRECISION, from, my_tag,
                    MPI_COMM_WORLD, my_status, ierror)
MPI_Recv(buffer,count,MPI_DOUBLE,from,my_tag,MPI_COMM_WORLD,&my_status);
MPI::COMM_WORLD.Recv(buffer,count,MPI_DOUBLE,from,my_tag,my_status);
```

- [21] Exercise 4(Local/Partial sum and full sum): **(Using six `MPI_***`.) Generalize your code to solve Exercise 3 to include the following:**
  - ⋆ **Each processing core should print its `local_sum` and the associated error term `|local_sum - math_local_sum|`, where `math_local_sum` is computed appropriate sum of the squares mathematical formula**
  - ⋆ **Each non-master processing core should send its `local_sum` and the master core should receive these private values and use these and its `local_sum` to compute `full_sum` =** $\sum_{i=1}^{N} a(i) = \sum_{i=1}^{N} i^2$
  - ⋆ **The master core should print `full_sum` and the final error term `|final_sum - math_final_sum|`, where `math_final_sum` is given by the sum of squares formula: `math_final_sum` =** $N(N+1)(2N+1)/6$
  - ⋆ **The master code in addition should print the total MPI walltime (required to complete all of the above tasks) and the total number of processing cores used to achieve complete the task**
  - ⋆ **Make your output easier to read, with appropriate information**

[21] M. Ganesh, MATH440/540, SPRING 2018

- [22]Exercise 4--continued:
  Test your Exercise 4 code with $N = 100,000$ and $P = 4, 8, 16, 32, 64$ and hence tabulate the speed-up $S_P = T_1/T_P$ and efficiency $E_P = S_P/P$, where T_1 and T_P denote the one-core (sequential) and P-core (parallel) wall-clock execution time of the code, respectively.

[22]M. Ganesh, MATH440/540, SPRING 2018