Parallel Programming
Basic MPI

Timothy H. Kaiser, Ph.D.
tkaiser2@nrel.gov

Slides: https://github.com/timkphd/slides

Talk Overview

- Background on MPI
- Documentation
- Hello world in MPI
- Basic communications
- Simple send and receive program

Background on MPI

- MPI - Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, & parallel programmers
  - Used to create parallel programs based on message passing
- Portable: one standard, many implementations
- Available on almost all parallel machines in C and Fortran
- Over 100 advanced routines but 6 basic

Unofficial Languages

- Subset available for R and Java
- Fairly complete implementation for Python
- Let me know if you are interested in these

Documentation

- MPI home page (contains the library standard): www.mcs.anl.gov/mpi
- Books
  - "MPI: The Complete Reference" by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press (also in Postscript and html)
  - "Using MPI" by Gropp, Lusk and Skjellum, MIT Press
- Tutorials

MPI Implementations

- Most parallel supercomputer vendors provide optimized implementations
- LAM
  - www.lam-mpi.org (deprecated)
- OpenMPI
  - www.open-mpi.org (default on Mio and RA)

MPI Implementations

- MPICH:
  - http://www-unix.mcs.anl.gov/mpi/mpich1/download.html
  - http://www.mcs.anl.gov/research/projects/mpich2/index.php
- MVAPICH & MVAPICH2
  - Infiniband optimized version of MPICH
  - http://mvapich.cse.ohio-state.edu/index.shtml

Key Concepts of MPI

- Used to create parallel programs based on message passing
  - Normally the same program is running on several different processors
  - Processors communicate using message passing
- Typical methodology:

```
start job on n processors
do i=1 to j
    each processor does some calculation
    pass messages between processor
end do
end job
```

Messages

- Simplest message: an array of data of one type.
- Predefined types correspond to commonly used types in a given language
  - MPI_REAL (Fortran), MPI_FLOAT (C)
  - MPI_DOUBLE_PRECISION (Fortran), MPI_DOUBLE (C)
  - MPI_INTEGER (Fortran), MPI_INT (C)
- User can define more complex types and send packages.

Communicators

- Communicator
  - A collection of processors working on some part of a parallel job
  - Used as a parameter for most MPI calls
  - MPI_COMM_WORLD includes all of the processors in your job
  - Processors within a communicator are assigned numbers (ranks) 0 to n-1
  - Can create subsets of MPI_COMM_WORLD

Include files

- The MPI include file
  - C: mpi.h
  - Fortran: mpif.h (a f90 module is a good place for this)
- Defines many constants used within MPI programs
- In C defines the interfaces for the functions
- Compilers know where to find the include files

Minimal MPI program

- Every MPI program needs these…
  - C version

```
/* the mpi include file */
#include <mpi.h>
   int nPEs,ierr,iam;
/* Initialize MPI */
   ierr=MPI_Init(&argc, &argv);
/* How many processors (nPEs) are there?*/
   ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What processor am I (what is my rank)? */
   ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
   ierr=MPI_Finalize();
```

In C MPI routines are functions and return an error value

Minimal MPI program

- Every MPI program needs these…
  - Fortran version

```
! MPI include file
      include 'mpif.h'
! The mpi module can be used for Fortran 90 instead of mpif.h
!     use mpi
      integer nPEs, ierr, iam
!  Initialize MPI
      call MPI_Init(ierr)
!  How many processors (nPEs) are there?
      call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
!  What processor am I (what is my rank)?
      call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
      ...
      call MPI_Finalize(ierr)
```

In Fortran, MPI routines are subroutines, and last parameter is an error value

Exercise 1 : Hello World

- Write a parallel "hello world" program
  - Initialize MPI
  - Have each processor print out "Hello, World" and its processor number (rank)
  - Quit MPI

Compiling

- Load module that points to your compilers
- For HP's version of MPI
  - module load mpt gcc/8.4.0
  - mpif77 mpif90
  - mpicc mpiCC
- For Intel compilers use
  - module load intel-mpi
  - mpiifort
  - mpiicc mpicpc
- Most MPI compilers are actually just scripts that call underlying Fortran or C compilers

Fortan and C examples

```
el2:mpi> cat helloc.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/*************************************************************
This is a simple hello world program. Each processor prints
name, rank, and total run size.
*************************************************************/
int main(int argc, char **argv)
{
    int myid,numprocs,resultlen;
    char myname[MPI_MAX_PROCESSOR_NAME] ;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(myname,&resultlen);
    printf("Hello from %s %d %d\n",myname,myid,numprocs);
    MPI_Finalize();
}
```

Fortan and C examples

```
el2:mpi> cat hellof.f90
!*********************************************
!  This is a simple hello world program. Each processor
!  prints out its name, rank and  number of processors
!  in the current MPI run.
!*********************************************
      program hello
      include "mpif.h"
      integer myid,numprocs,ierr,nlength
      character (len=MPI_MAX_PROCESSOR_NAME):: myname
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      call MPI_Get_processor_name(myname,nlength,ierr)
      write (*,*) "Hello from ",trim(myname)," # ",myid," of ",numprocs
      call MPI_FINALIZE(ierr)
      stop
      end
el2:mpi>
el2:mpi>
```

Building

```
el2:mpi> ml intel-mpi
el2:mpi>
el2:mpi> mpiicc helloc.c -o example
el2:mpi> mpiifort hellof.f90 -o example
el2:mpi>
```

Or

```
el2:mpi> module load mpt gcc/8.4.0
el2:mpi>
el2:mpi> mpicc helloc.c -o example
el2:mpi> mpif90 hellof.f90 -o example
el2:mpi>
```

Running

- Most often you will use a batch system
- Write a batch script file.
- Use the command ~~**mpiexec**~~ or ~~**mpirun**~~ to actually start the program. **srun**
- You must tell the system how many copies to run
- On some systems you must tell where to run the program

A very simple Slurm Script

```
#!/bin/bash
#SBATCH --job-name="flow"
#SBATCH --nodes=1
#SBATCH --export=ALL
#SBATCH --oversubscribe
#SBATCH --time=00:10:00
#SBATCH --partition=debug
#SBATCH --account=hpcapps

module purge
ml intel-mpi

srun -n 4 ./example
```

To run...

```
el2:mpi>
el2:mpi> sbatch sim
Submitted batch job 7068634
el2:mpi>
```

Output

```
el2:mpi> cat slurm-7068634.out
Hello from r1i7n35 0 4
Hello from r1i7n35 1 4
Hello from r1i7n35 2 4
Hello from r1i7n35 3 4
el2:mpi>
```

Basic Communication

- Data values are transferred from one processor to another
  - One processor sends the data
  - Another receives the data
- Synchronous
  - Call does not return until the message is sent or received
- Asynchronous

Synchronous Send

- C
  - MPI_Send(&buffer, count ,datatype, destination, tag,communicator);
- Fortran
  - Call MPI_Send(buffer, count, datatype, destination,tag,communicator, ierr)
- Call blocks until message on the way

Call MPI_Send(buffer, count, datatype, destination, tag, communicator, ierr)

- **Buffer**: The data array to be sent
- **Count** : Length of data array (in elements, 1 for scalars)
- **Datatype** : Type of data, for example : MPI_DOUBLE_PRECISION, MPI_INT, etc
- **Destination** : Destination processor number (within given communicator)
- **Tag** : Message type (arbitrary integer)
- **Communicator** : Your set of processors
- **Ierr** : Error return (Fortran only)

Synchronous Receive

- C
  - MPI_Recv(&buffer,count, datatype, source, tag, communicator, &status);
- Fortran
  - Call MPI_ RECV(buffer, count, datatype, source,tag,communicator, status, ierr)
- Call blocks the program until message is in buffer
- Status - contains information about incoming message
  - C
    - MPI_Status status;
  - Fortran
    - Integer status(MPI_STATUS_SIZE)

Call MPI_Recv(buffer, count, datatype, source, tag, communicator, status, ierr)

- **Buffer**: The data array to be received
- **Count** : Maximum length of data array (in elements, 1 for scalars)
- **Datatype** : Type of data, for example : MPI_DOUBLE_PRECISION, MPI_INT, etc
- **Source** : Source processor number (within given communicator)
- **Tag** : Message type (arbitrary integer)
- **Communicator** : Your set of processors
- **Status**: Information about message
- **Ierr** : Error return (Fortran only)

Exercise 2 : Basic Send and Receive

- Write a parallel program to send & receive data
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive an integer from processor 0
  - Both processors print the data
  - Quit MPI

Send and Recv in MPI

```
el2:mpi> cat f_ex01.f90
      module fmpi
!DEC$ NOFREEFORM
      include "mpif.h"
!DEC$ FREEFORM
      end module
!*************************************************************
!  This is a simple send/receive program in MPI
!  Processor 0 sends an integer to processor 1,
!  while processor 1 receives the integer from proc. 0
!*************************************************************
      program hello
      use fmpi
!     include "mpif.h"
      integer myid, ierr,numprocs
      integer tag,source,destination,count
      integer buffer
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      tag=1234
      source=0
      destination=1
      count=1
      if(myid .eq. source)then
         buffer=5678
         Call MPI_Send(buffer, count, MPI_INTEGER,destination,&
          tag, MPI_COMM_WORLD, ierr)
         write(*,*)"processor ",myid," sent ",buffer
      endif
      if(myid .eq. destination)then
         Call MPI_Recv(buffer, count, MPI_INTEGER,source,&
          tag, MPI_COMM_WORLD, status,ierr)
         write(*,*)"processor ",myid," got ",buffer
      endif
      call MPI_FINALIZE(ierr)
      stop
      end
```

Send an Recv in MPI

```
el2:mpi> cat c_ex01.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

/*************************************************************
This is a simple send/receive program in MPI
*************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;
    if(myid == source){
      buffer=5678;
      MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
      printf("processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d  got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```

```
el2:mpi> cat slurm-7068778.out
 processor             0  sent        5678
 processor             1  got         5678
el2:mpi>
```

Summary

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- Well over 100 "Advanced Calls"
- The 6 basic calls in MPI are:

```
– MPI_INIT( ierr )
– MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
– MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
– MPI_Send(buffer, count,MPI_INTEGER,destination, tag, MPI_COMM_WORLD, ierr)
– MPI_Recv(buffer, count, MPI_INTEGER,source,tag, MPI_COMM_WORLD, status,ierr)
– MPI_FINALIZE(ierr)
```