Parallel Programming Basic MPI

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

Examples and Slides

Examples:

```
git clone https://github.com/timkphd/examples
cd examples/mpi
```

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
- Also used to create libraries.

Unofficial Languages

- Subsets available for R and Java
- Fairly complete implementation for Python
- Let me know if you are interested in these
- I have not looked at Julia

Documentation

- MPI home page (contains the library standard) https://www.mpi-forum.org
- 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
  - many online, just do a search

MPI Implementations

- Most parallel supercomputer vendors provide optimized implementations
- Intel
- HP/SGI/Cray - mpt
- IBM

Open Source MPI Implementations

- MPICH:
  - https://www.mpich.org
- Mvapich
  - http://mvapich.cse.ohio-state.edu
- OpenMPI
  - www.open-mpi.org

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
```

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

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.

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

Fortran 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();
}
```

Fortran 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>
```

Compiling

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

Compiling IntelMPI and MPT

```
el2:mpi> ml intel-mpi
el2:mpi>
el2:mpi> mpiicc helloc.c -o helloc
el2:mpi> mpiifort hellof.f90 -o hellof
el2:mpi> #### or ####
el2:mpi> mpicc helloc.c -o helloc
el2:mpi> mpif90 hellof.f90 -o hellof
```

Or

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

Compiling OpenMPI

- Most MPI compilers are actually just scripts that call underlying Fortran or C compilers
- Load module that points to your compilers
- Openmpi
  - module load openmpi
  - mpif77 mpif90
  - mpicc mpiCC mpic++ mpicxx
- This will give you gcc/gfortran as the backend compilers
- To get Intel backend compilers
  - export OMPI_FC=ifort
  - export OMPI_CC=icc
  - export OMPI_CXX=icpc

makefile

```
el1:mpi> ml intel-mpi
el1:mpi> make
mpicc   c_ex00.c -o c_ex00
mpicc   c_ex01.c -o c_ex01
mpicc   c_ex02.c -o c_ex02
mpicc   c_ex03.c -o c_ex03
mpicc   c_ex04.c -o c_ex04
mpicc   c_ex05.c -o c_ex05
mpicc   c_ex06.c -o c_ex06
mpicc   c_ex07.c -o c_ex07
mpicc   c_ex08.c -o c_ex08
mpicc   c_ex09.c -o c_ex09
mpicc   c_ex10.c -o c_ex10
mpicc   c_ex11.c -o c_ex11
mpicc   c_ex12.c -o c_ex12
mpicc   c_ex13.c -o c_ex13
c_ex13.c: In function 'main':
c_ex13.c:34:23: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
  mpi_err = MPI_Gather((void*)mysize,1,MPI_INT,
                       ^
mpicc   helloc.c -o helloc
mpif90   f_ex00.f90 -o f_ex00
mpif90   f_ex01.f90 -o f_ex01
mpif90   f_ex02.f90 -o f_ex02
mpif90   f_ex03.f90 -o f_ex03
mpif90   f_ex04.f90 -o f_ex04
mpif90   f_ex05.f90 -o f_ex05
mpif90   f_ex06.f90 -o f_ex06
mpif90   f_ex07.f90 -o f_ex07
mpif90   f_ex08.f90 -o f_ex08
mpif90   f_ex09.f90 -o f_ex09
mpif90   f_ex10.f90 -o f_ex10
mpif90   f_ex11.f90 -o f_ex11
mpif90   f_ex12.f90 -o f_ex12
mpif90   f_ex13.f90 -o f_ex13
mpif90   hellof.f90 -o hellof
el1:mpi>
```

Running

- Most often you will use a batch system
- Write a batch script file.
- Use the command ~~**mpiexec**~~ or ~~**mpirun**~~ **srun** to actually start the program. 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 - "sim"

```
#!/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 ./helloc
```

The run...

```
el1:mpi> sbatch --account=hpcapps sim
Submitted batch job 8182395
el1:mpi> cat slurm-8182395.out
 Hello from r1i7n35 #      2  of      4
 Hello from r1i7n35 #      0  of      4
 Hello from r1i7n35 #      1  of      4
 Hello from r1i7n35 #      3  of      4
el1:mpi>
```

A useful example - "doeagle"

```
#!/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

if [ -z ${EXE+x} ]; then export EXE=c_ex00 ; fi

echo "running " $EXE
srun -n 4 ./$EXE
```

You can set the program to run with and export before you submit the run

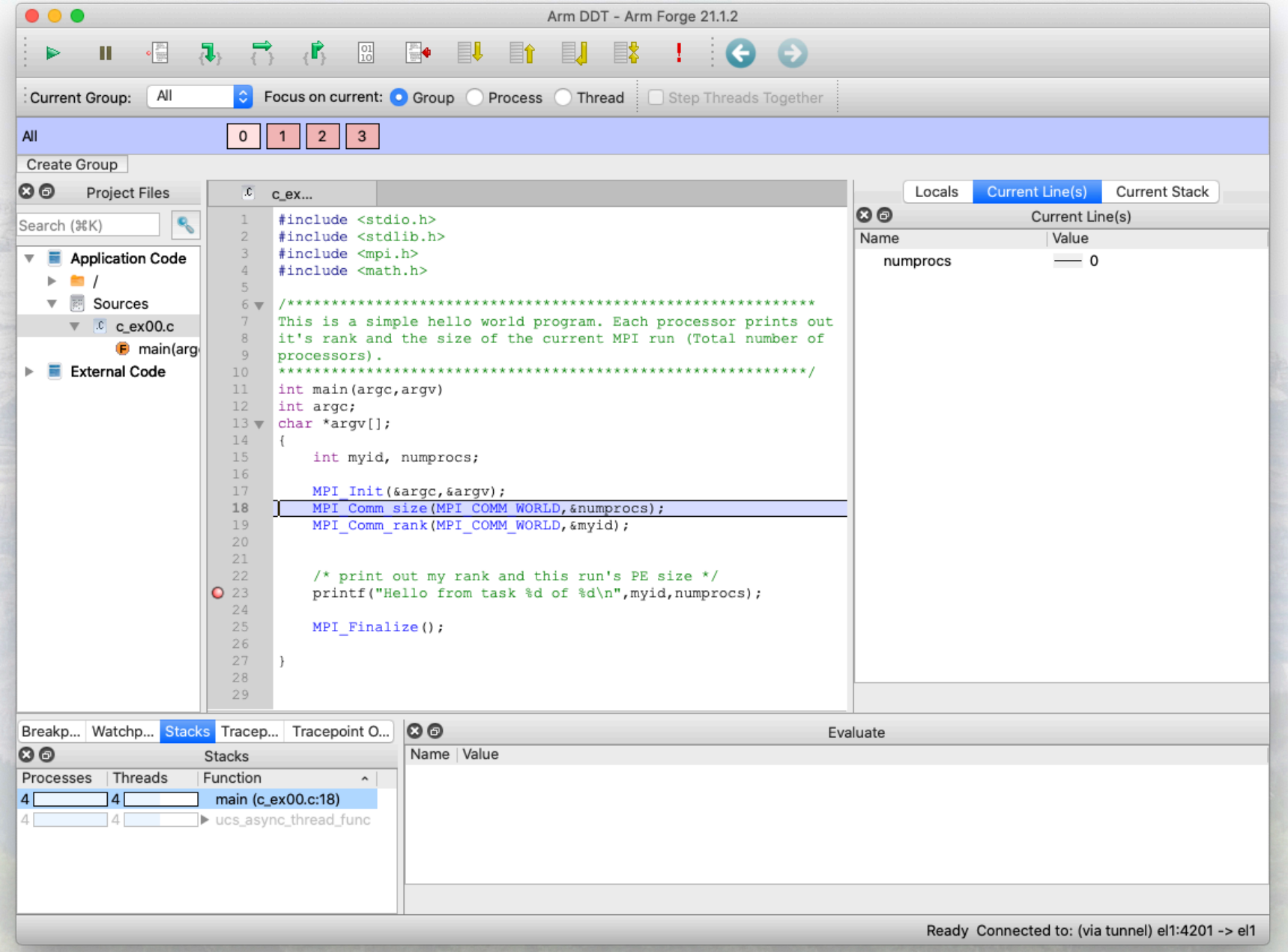
```
el1:mpi> export EXE=f_ex02
el1:mpi> sbatch --account=hpcapps doeagle
Submitted batch job 8182393
el1:mpi> cat *8182393*
running  f_ex02
 Hello from fortran process:           3  Numprocs is           4
 Hello from fortran process:           1  Numprocs is           4
 Hello from fortran process:           2  Numprocs is           4
 Hello from fortran process:           0  Numprocs is           4
 getting            1
 i=         100
el1:mpi>
```

The run...

Slight Digression

- One way you can understand what is going on in a MPI application is to look at it running in a debugger
- We have ARM/ddt
- We also have ARM/map and ARM/Profile for profiling
- Let me know if interested

ARM ddt


Arm DDT - Arm Forge 21.1.2
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
All 0 1 2 3
Create Group
Project Files
Search (⌘K)
Application Code
/
Sources
c_ex00.c
main(arg
External Code
c_ex...
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
/***
This is a simple hello world program. Each processor prints out
it's rank and the size of the current MPI run (Total number of
processors).
***/
int main(argc,argv)
int argc;
char *argv[];
{
int myid, numprocs;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
/* print out my rank and this run's PE size */
printf("Hello from task %d of %d\n",myid,numprocs);
MPI_Finalize();
}
Locals
Current Line(s)
Current Stack
Current Line(s)
Name
Value
numprocs
0
Breakp...
Watchp...
Stacks
Tracep...
Tracepoint O...
Stacks
Processes
Threads
Function
4
4
main (c_ex00.c:18)
4
4
ucs_async_thread_func
Evaluate
Name
Value
Ready
Connected to: (via tunnel) el1:4201 -> el1

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
  - Call indicates a start of send or receive, and another call is made to determine if finished

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)
```

MPI is not limited to huge HPC

- Can be built on laptops and desktops
  - Might be useful for testing
  - Might actually get good performance
- I have it on my laptop & high end desktop
- I have it running on my home 4 node Raspberry Pi cluster
  - https://www.raspberrypi.org
  - < $100/node

Unofficial Languages

- Subsets available for R and Java
- Fairly complete implementation for Python
- Let me know if you are interested in these
- I have not looked at Julia

Java

- Comes in two versions OpenMPI and Intel
- They are not compatible even at the source (API) level
  - cd ../java
  - diff Hello.java Ihello.java

```
...
<    int myrank = MPI.COMM_WORLD.getRank();
<    int size = MPI.COMM_WORLD.getSize() ;
>    mpi.Comm  comm = mpi.Comm.WORLD;
>    int myrank = comm.getRank();
>    int size = comm.getSize();

<       MPI.COMM_WORLD.send(message, tosend, MPI.INT, 1, tag);
>       mpi.PTP.send(message,10,mpi.Datatype.INT,1,tag,comm);

<       MPI.COMM_WORLD.recv(message, toget, MPI.INT, 0, tag);
>       mpi.PTP.recv(message,10,mpi.Datatype.INT,0,tag,comm);
```

Build Rmpi & MPI4py

I used a Conda environment that already had R and Python. Then I compiled using IntelMPI, not OpenMPI

```
cd mixedlang

ml conda
conda activate plex1
which python
which R
ml comp-intel/2020.1.217
ml intel-mpi/2020.1.217
which mpicc
```

```
curl --insecure https://cran.r-project.org/src/contrib/Rmpi_0.6-9.2.tar.gz -o Rmpi.tar.gz

export TYPE=OPENMPI
export MY_MPI_PATH=/nopt/nrel/apps/compilers/intel/2020.1.217/impi/2019.7.217/intel64

R CMD INSTALL --configure-args="\
--with-Rmpi-include='$MY_MPI_PATH/include' \
--with-Rmpi-libpath='$MY_MPI_PATH/lib/release' \
--with-mpi='$MY_MPI_PATH/bin/mpicc' \
--with-Rmpi-type='$TYPE'"  Rmpi.tar.gz


pip install mpi4py
```

It is possible(sometimes) to mix languages

```
(/home/tkaiser2/.conda-envs/plex1) el3:mixedlang> cat together
#!/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
#
##r_ex01c.R does not work with openmpi and other programs
#j_ex01c.java only works with openmpi
#ij_ex01c.java is for Intel MPI and works with all others
#
#
export PATH=/nopt/nrel/apps/openmpi/4.1.0-gcc-8.4.0-j15/jdk-15.0.2/bin:$PATH
module purge
ml conda
conda activate plex1
which python
which R
ml comp-intel/2020.1.217
ml intel-mpi/2020.1.217
which mpicc

mpif90 f_ex01c.f90 -o f_ex01c ; rm -rf fmpi.mod
mpicc c_ex01c.c -o c_ex01c

#for source in f_ex01c  c_ex01c r_ex01c.R  P_ex01c.py  "java ij_ex01c.java" ; do
#for source in f_ex01c  c_ex01c r_ex01c.R  P_ex01c.py  ; do
for source in f_ex01c  c_ex01c r_ex01c.R  P_ex01c.py   ; do
   echo DRIVER: $source
#mpiexec -n 1 $source  2 5 : -n 1 ./f_ex01c : -n 1 ./c_ex01c : -n 1 ./r_ex01c.R  : -n 1  ./P_ex01c.py  : -n 1 java ij_ex01c.java
| egrep "got"
   echo 0 ./$source     > mapfile
   echo 1 ./f_ex01c    >> mapfile
   echo 2 ./c_ex01c    >> mapfile
   echo 3 ./r_ex01c.R  >> mapfile
   echo 4 ./P_ex01c.py >> mapfile
   echo 5 java ./ij_ex01c.java >> mapfile
   #srun -n 5 --multi-prog mapfile | grep got
   srun --partition=debug --time=00:10:00 -n 6 --multi-prog mapfile | grep got
   echo " "
done
```

```
DRIVER: f_ex01c
Fortran processor    1   got     4678    3678    2678    1678
Fortran processor    1   got     4679    3679    2679    1679
Fortran processor    1   got     4680    3680    2680    1680
C processor 2  got 4678  3678  2678  1678
C processor 2  got 4679  3679  2679  1679
C processor 2  got 4680  3680  2680  1680
Java processor 5  got 4678 3678 2678 1678
Java processor 5  got 4679 3679 2679 1679
Java processor 5  got 4680 3680 2680 1680
[1] "R processor    3 got  4678  3678  2678  1678"
[1] "R processor    3 got  4679  3679  2679  1679"
[1] "R processor    3 got  4680  3680  2680  1680"
Python processor  4  got  [4678 3678 2678 1678]
Python processor  4  got  [4679 3679 2679 1679]
Python processor  4  got  [4680 3680 2680 1680]

DRIVER: c_ex01c
C processor 2  got 1  101  201  301
Fortran processor    1   got       1     101     201     301
Fortran processor    1   got       2     102     202     302
Fortran processor    1   got       3     103     203     303
C processor 2  got 2  102  202  302
C processor 2  got 3  103  203  303
Java processor 5  got 1 101 201 301
Java processor 5  got 2 102 202 302
Java processor 5  got 3 103 203 303
[1] "R processor    3 got  1  101  201  301"
[1] "R processor    3 got  2  102  202  302"
[1] "R processor    3 got  3  103  203  303"
Python processor  4  got  [  1 101 201 301]
Python processor  4  got  [  2 102 202 302]
Python processor  4  got  [  3 103 203 303]
```

```
DRIVER: r_ex01c.R
Fortran processor    1   got       3       5       7       9
C processor 2  got 3  5  7  9
Fortran processor    1   got       4       6       8      10
C processor 2  got 4  6  8  10
Fortran processor    1   got       5       7       9      11
C processor 2  got 5  7  9  11
[1] "R processor    3 got  3  5  7  9"
[1] "R processor    3 got  4  6  8  10"
[1] "R processor    3 got  5  7  9  11"
Java processor 5  got 3 5 7 9
Java processor 5  got 4 6 8 10
Java processor 5  got 5 7 9 11
Python processor  4  got  [3 5 7 9]
Python processor  4  got  [ 4  6  8 10]
Python processor  4  got  [ 5  7  9 11]

DRIVER: P_ex01c.py
Fortran processor    1   got     6679    7679    8679    9679
C processor 2  got 6679  7679  8679  9679
Fortran processor    1   got     6680    7680    8680    9680
C processor 2  got 6680  7680  8680  9680
Fortran processor    1   got     6681    7681    8681    9681
C processor 2  got 6681  7681  8681  9681
Java processor 5  got 6679 7679 8679 9679
Java processor 5  got 6680 7680 8680 9680
Java processor 5  got 6681 7681 8681 9681
[1] "R processor    3 got  6679  7679  8679  9679"
[1] "R processor    3 got  6680  7680  8680  9680"
[1] "R processor    3 got  6681  7681  8681  9681"
Python processor  4  got  [6679 7679 8679 9679]
Python processor  4  got  [6680 7680 8680 9680]
Python processor  4  got  [6681 7681 8681 9681]
```