MPI

Luciano Laratelli

PRESENTATION GOALS

1. Explaining what MPI is
2. Overview of fundamental concepts
3. Installing, compiling, and running MPI
4. Code samples

MPI; WHAT IS IT?

MPI IS A *SPECIFICATION*

- MPI Forum defines which operations must be implemented with an LIS (Language Independent Specification)
- Must be implemented for C, Fortran -- many other APIs exist (Python, C++, C#, etc.)
- Over 20 notable implementations exist

MPI VS CUDA

- MPI and CUDA both implement the SPMD programming model
- CPU vs GPU
- MPI can take advantage of distributed (non-shared) memory
- MPI is hardware-independent

MPI VERSIONS

1. MPI-1: original version, static runtime
2. MPI-2: parallel I/O, dynamic processes
3. MPI-3: non-blocking operations, other extensions

POPULAR IMPLEMENTATIONS

- MPICH - initial implementation from ANL, implements MPI-3
- Open MPI - merger between three well-known implementations, implements MPI-3
- Boost.MPI - C++ interface, implements MPI-1; easy to interface with other MPI implementations
- Intel MPI Library - proprietary extension of MPICH optimized for Intel processors

INSTALLING COMPILING RUNNING

GETTING MPI

- System package manager, under names like:
  - `open-mpi`
  - `mpich`
- USF CIRCE:
  - `module load mpi/openmpi/1.4.1`
  - `module load mpi/mpich2/3.1.4`

THE MPI COMPILER

- Just like `nvcc`, MPI implementations provides compiler wrappers which compiles source code with the proper libraries:
  - `mpicc` (C)
  - `mpic++/mpicxx` (C++)
  - `mpif/mpif90/mpifort` (Fortran)

RUNNING MPI

- Use mpiexec
- –n flag specifies number of processes
- mpiexec –n 4 ./a.out <programArgs>

FUNDAMENTALS: COMMUNICATORS

COMMUNICATORS

- A communicator is an object containing every MPI process -- it handles communication between processes
- Each process in a communicator is numbered -- this is called the process' *rank*
- Default communicator (`MPI_COMM_WORLD`) fine for many use cases, but user can create communicators at runtime as needed

```c
#include <mpi.h>

int main(int argc, char ** argv){
  MPI_Init(&argc, &argv);
  //ALL OTHER MPI CALLS GO HERE
  MPI_Finalize();
}
```

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

int main(int argc, char ** argv) {
  MPI_Init(&argc, &argv);

  int worldSize;
  int rank;

  MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello from rank %d out of %d processors\n", rank,
worldSize);

  MPI_Finalize();
}
```

```
bash-5.0$ mpicc hello.c -o hello
bash-5.0$ mpirun -np 4 ./hello
Hello from rank 2 out of 4 processors
Hello from rank 0 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 3 out of 4 processors
```

```
(Sun Jul-7 10:13:40P)-(luciano:~/Dropbox/
> mpiexec -np 4 ./hello
Hello from rank 2 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 0 out of 4 processors
Hello from rank 3 out of 4 processors
(Sun Jul-7 10:13:46P)-(luciano:~/Dropbox/
> mpiexec -np 4 ./hello
Hello from rank 0 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 3 out of 4 processors
Hello from rank 2 out of 4 processors
(Sun Jul-7 10:13:47P)-(luciano:~/Dropbox/
> mpiexec -np 4 ./hello
Hello from rank 2 out of 4 processors
Hello from rank 0 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 3 out of 4 processors
(Sun Jul-7 10:13:48P)-(luciano:~/Dropbox/
> mpiexec -np 4 ./hello
Hello from rank 0 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 2 out of 4 processors
Hello from rank 3 out of 4 processors
```

FUNDAMENTALS: MPI DATATYPES

MPI DATATYPES

- MPI datatypes are implementation-provided macros
- They denote the type of data that is being handled by an MPI function when it is called
- User can define custom datatypes at runtime

WHY DO MPI DATATYPES EXIST?

- Nodes can be split between machines with different architectures
- MPI functions are designed to know what they are working with
- Allows for implementation to deal with type details

COMMON MPI DATATYPES

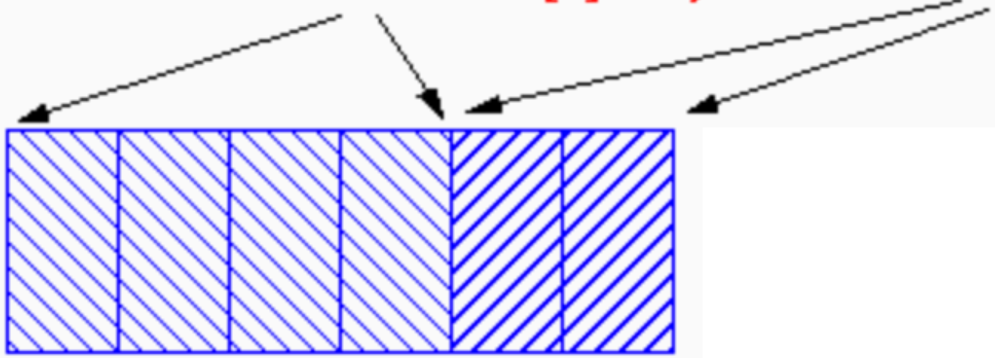
MPI Datatype	C Type
`MPI_CHAR`	`char`
`MPI_INT`	`int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`

ROLLING YOUR OWN

- **Derived** data types are user-defined data structures that are sequences of existing MPI datatypes
- Several variants:
  - Vector
  - Struct
  - Indexed

MPI_ Type_struct

```
typedef  struct  { float x, y, z, velocity;   int  n, type; }  Particle;
Particle  particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2;  oldtypes[0] = MPI_FLOAT;    oldtypes[1] = MPI_INT
            offsets[0] = 0;             offsets[1] = 4 * extent;
            blockcounts[0] = 4;         blockcounts[1] = 2;
```



```c
typedef struct {
    float x, y, z, velocity;
    int n, type;
} Particle;

MPI_Datatype particletype;  // variable that will hold our new type
MPI_Datatype oldtypes[2];   // types from which we derive the new type

int blockcounts[2];   // amount of variables of a certain type
MPI_Aint offsets[2];  // after what amount of bytes does the current type start
MPI_Aint extent;      // size of the MPI datatype

offsets[0] = 0;             // floats start at byte 0 of the type
oldtypes[0] = MPI_FLOAT;    // first block is MPI_FLOATs
blockcounts[0] = 4;         // the block contains 4 MPI_FLOATs

MPI_Type_extent(MPI_FLOAT, &extent);  // get size of MPI_Float

offsets[1] = 4 * extent;  // ints start at byte 4 * extent of the type
oldtypes[1] = MPI_INT;    // second block is MPI_INTs
blockcounts[1] = 2;       // this block contains 2 MPI_INTs

MPI_Type_struct(2,                // number of blocks
                blockcounts,   // elements in each block [array]
                offsets,       // byte displacement from 0 for each block [array]
                oldtypes,      // type of eleements in each block [array]
                &particletype  // new data type);
MPI_Type_commit(&particletype);

// free datatype when done using it
MPI_Type_free(&particletype);
```

FUNDAMENTALS:
POINT-TO-POINT OPERATIONS

P2P OPERATIONS

- Direct communication between two ranks: one sends and the other receives
- Serial or asynchronous variants

(NON)BLOCKING CALLS

- **Blocking** calls do not return until their underlying process has executed completely
  - `MPI_Send`, `MPI_Recv`, etc.
- **Non-blocking** calls return immediately (can lead to performance improvements)
  - `MPI_ISend`, `MPI_IRecv`, etc.

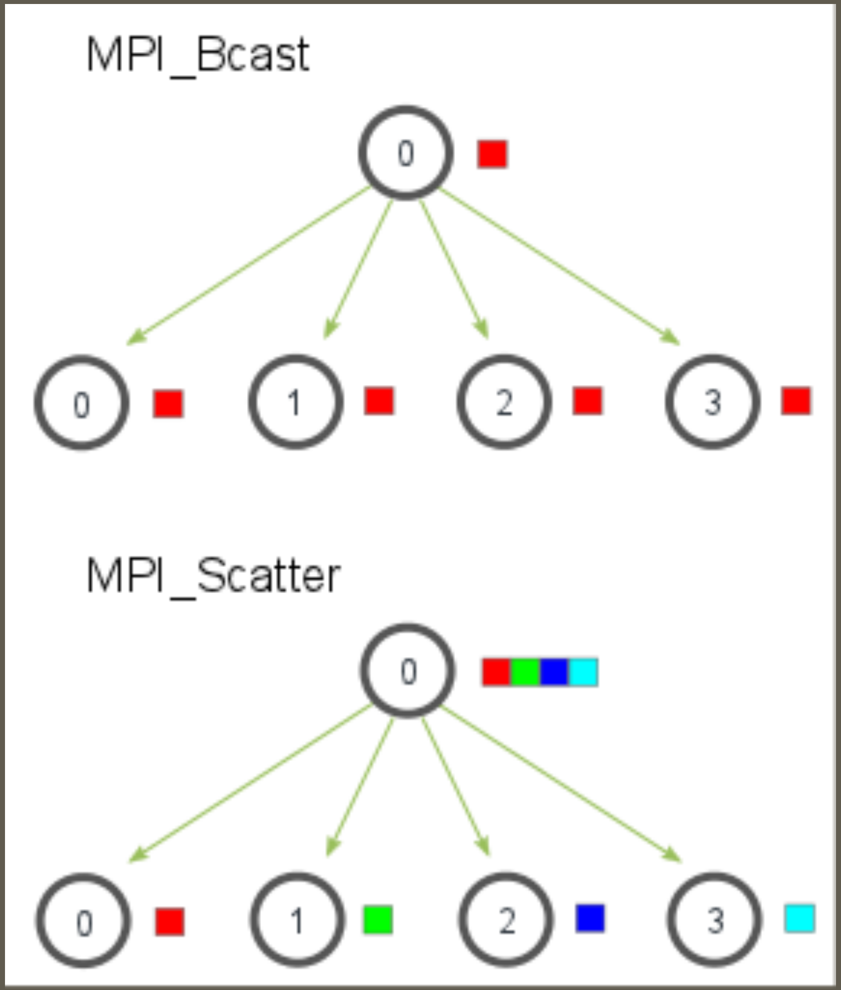
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  MPI_Init(&argc, &argv);
  int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int size; MPI_Comm_size(MPI_COMM_WORLD, &size);

  int number = rank * 72;
  MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  if(rank == 0){
    for(int i = 0; i < size; i++){
      MPI_Recv(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("received %d from process with rank %d\n", number, i);
    }
  }
  MPI_Finalize();
  return 0;
}
```

FUNDAMENTALS: COLLECTIVE OPERATIONS

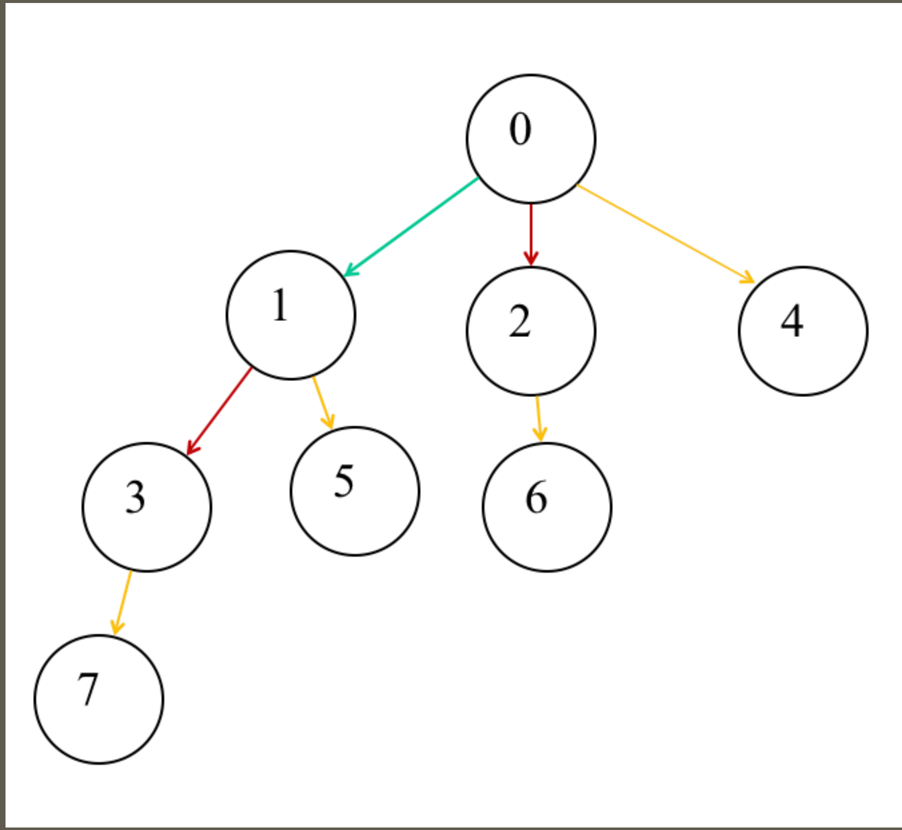
COLLECTIVE OPERATIONS

- Moving data from one rank to many ranks, or vice-versa
- Semantically equivalent to doing many `MPI_Send`s and `MPI_RECV`s, but faster via implementation tricks

BROADCAST / SCATTER

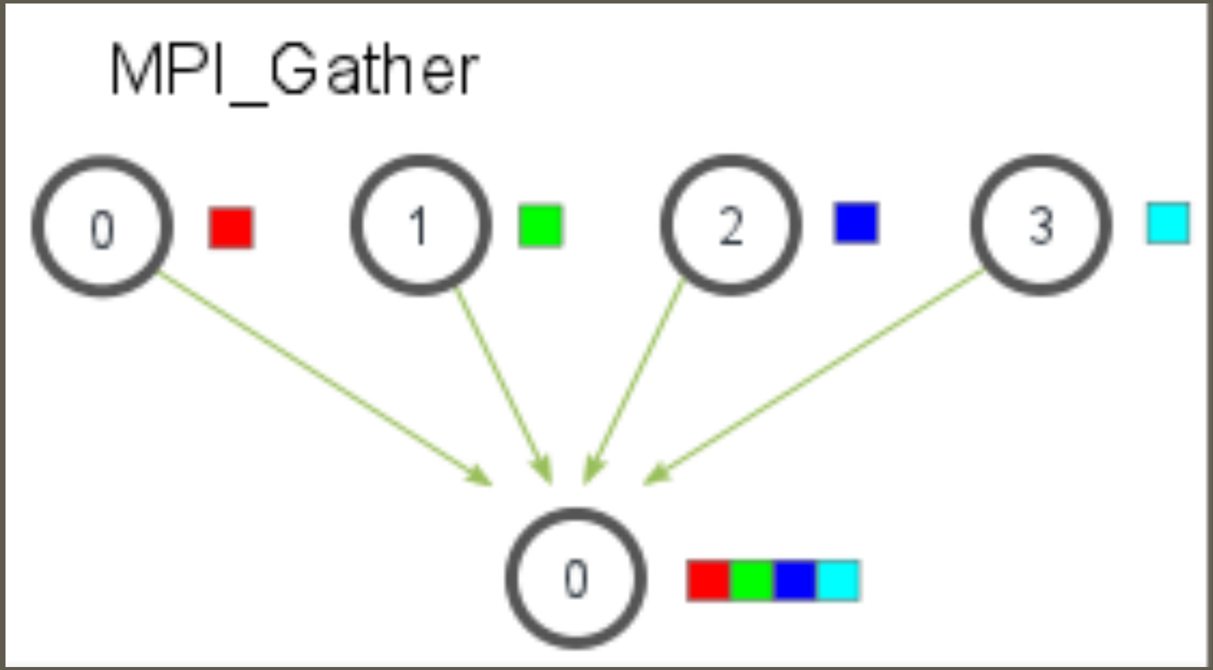


IMPLEMENTATION: BROADCAST



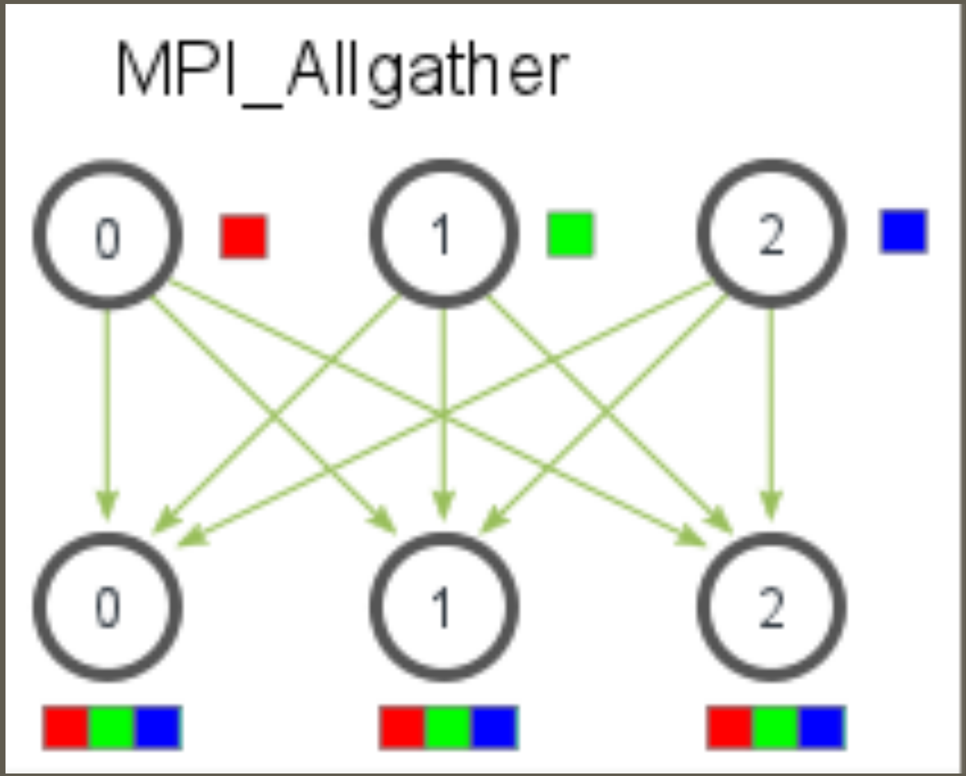
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int rank, buf;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank == 0) {
    buf = 777;
  }

  printf("rank %d, buf is %d\n", rank, buf);
  // everyone calls bcast, data is taken from root and ends up in everyone's buf
  MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);

  printf("rank %d, buf is %d\n", rank, buf);

  MPI_Finalize();
  return 0;
}
```

GATHER



```c
int *localA = (int*)malloc(sizeof(int) * vecSize / size);
int *localB = (int*)malloc(sizeof(int) * vecSize / size);
int *localResult = (int*)malloc(sizeof(int) * vecSize / size);
int *a = NULL, *b = NULL, *result = NULL;


if(rank == 0) {
  a = (int*)malloc(vecSize * sizeof(int));
  b = (int*)malloc(vecSize * sizeof(int));
  result = (int*)malloc(vecSize * sizeof(int));
  for(int i = 0; i < vecSize; i++){
    a[i] = i;
    b[i] = i * 2;
  }
}

MPI_Scatter(a, vecSize/size, MPI_INT, localA, vecSize/size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter(b, vecSize/size, MPI_INT, localB, vecSize/size, MPI_INT, 0, MPI_COMM_WORLD);

vecAdd(localResult, localA, localB, vecSize/size);

MPI_Gather(localResult, vecSize/size, MPI_INT, result, vecSize/size, MPI_INT, 0,
MPI_COMM_WORLD);
```

ALL-GATHER



COMMON ERRORS

ASKING FOR TOO MANY CORES

- You can request only up to the number of *physical* cores available

```
--------------------------------------------------------------------------
bash-5.0$ mpirun -np 8 ./hello
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 8
slots that were requested by the application:

  ./hello

Either request fewer slots for your application, or make more slots
available for use.
```

NOT CALLING MPI_FINALIZE

- MPI communicators must be destroyed before the program exits (think of it like the malloc/free idiom)

```
bash-5.0$ mpirun ./hello
Hello from rank 1 out of 4 processors
Hello from rank 2 out of 4 processors
Hello from rank 3 out of 4 processors
Hello from rank 0 out of 4 processors
--------------------------------------------------------------------------
mpirun has exited due to process rank 3 with PID 0 on
node Ls-MacBook-Pro exiting improperly. There are three reasons this could occur:

1. this process did not call "init" before exiting, but others in
the job did. This can cause a job to hang indefinitely while it waits
for all processes to call "init". By rule, if one process calls "init",
then ALL processes must call "init" prior to termination.

2. this process called "init", but exited without calling "finalize".
By rule, all processes that call "init" MUST call "finalize" prior to
exiting or it will be considered an "abnormal termination"

3. this process called "MPI_Abort" or "orte_abort" and the mca parameter
orte_create_session_dirs is set to false. In this case, the run-time cannot
detect that the abort call was an abnormal termination. Hence, the only
error message you will receive is this one.

This may have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).

You can avoid this message by specifying -quiet on the mpirun command line.
--------------------------------------------------------------------------
```

MISMATCHED SEND/RECV

```
#include <mpi.h>

int main(int argc, char ** argv){
  MPI_Init(&argc, &argv);
  int size; MPI_Comm_size(MPI_COMM_WORLD, &size);
  int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Recv(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, 0);
  MPI_Finalize();
}
```

BCAST/GATHER FROM ONE RANK

```
int* localBuf = malloc(sizeof(int) * 4);
int *buf = NULL, *res = NULL;
if (rank == 0) {
    buf = malloc(sizeof(int) * 16);
    res = malloc(sizeof(int) * 16);
    for (int i = 0; i < 16; i++) {
        buf[i] = i + 1;
    }
}
MPI_Scatter(buf, 4, MPI_INT, localBuf, 4, MPI_INT, 0, MPI_COMM_WORLD);
for (int i = 0; i < 4; i++) {
    localBuf[i] *= 2;
}
if (rank == 0) {
    MPI_Gather(localBuf, 4, MPI_INT, res, 4, MPI_INT, 0, MPI_COMM_WORLD);
    for (int i = 0; i < 16; i++) {
        printf("%d ", res[i]);
    }
}
```

MPI ALTERNATIVES

- POSIX Threads
- OpenMP
- PNL's ComEx
- GASnet

QUESTIONS

Do you have any?

RESOURCES

https://computing.llnl.gov/tutorials/mpi/

https://www.mpich.org/static/docs/v3.1/

https://www.open-mpi.org/doc/current/

https://mpitutorial.com/tutorials/