Message Passing Interface (MPI)

Jingke Li

Portland State University

What is MPI?

A message-passing library specification for explicitly programming message-passing systems.

- Fully featured
  - point-to-point, collective, and one-sided communications
  - I/O routines and profiling interface
- Multiple language bindings
  - C, C++ and Fortran
- Available on many platforms
  - Production grade implementations on supercomputers Don't use OMPI
  - Open source implementations (*e.g.* Open MPI) on Linux and Windows
- Has been stable for many years: very stable
  - MPI 2.0 was released in 1996, and it lasted 16 years
  - MPI 3.0 was released in Sept. 2012 (Manual 852 pages)

Different features of mpi:

Features of MPI

- *Point-to-Point Communications*
  - Structured buffers and derived data types, heterogeneity
  - many modes: blocking vs non-blocking vs synchronous, ready vs buffered
- *Collective Communications*
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology
  - Built-in support for grids and graphs
- *One-Sided Communications* (Supported but as easy to use as other)
  - Allow remote memory access
- *Others* Profiling hooks.
  - Message security, thread safety message security.
  - Profiling hooks, error control

Communicators combine context and group for

MPI for Beginners

One need not master all parts of MPI to use it. These six functions allow you to write many programs:

- `MPI_Init` — starting MPI } required.
- `MPI_Finalize` — exiting MPI
- `MPI_Comm_size` — the number of processes } required
- `MPI_Comm_rank` — the id of *this* process
- `MPI_Send` — sending a message } can't run mpi without mpi
- `MPI_Recv` — receiving a message

"Hello world!" in MPI

- C Version:

```c
#include "mpi.h"
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  MPI_Finalize();
  return 0;
}
```

- C++ Version:

```cpp
#include "mpi++.h"
int main(int argc, char **argv) {
  MPI::Init(argc, argv);
  cout << "Hello world!" << endl;
  MPI::Finalize();
  return 0;
}
```

`MPI_Init()` and `MPI_Finalize()` must be included in every MPI program.

"Hello world!" in MPI (Version 2)

```c
#include "mpi.h"
int main(int argc, char **argv) {
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world! I'm %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Communication in MPI takes place with respect to *communicators*.

- `MPI_COMM_WORLD` is the default communicator, it contains all MPI processes. User can define other communicators.
- `MPI_Init()` and `MPI_Finalize()` must be called by *all* processes.

Basic Send/Receive

to simplify mapping between different languages types

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
             int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype type, int src,
             int tag, MPI_Comm comm, MPI_Status *st)
```

All required parameters:

- buf — initial address of send/receive buffer (holds the data to send out)
- count, type — number of and type of elements to be sent/received (size of the buffer)
- dest, src — rank of destination and source processes; a special value for src is MPI_ANY_SOURCE (can use a wild card to pick multiple sources/dst)
- tag — message tag
- comm — communicator (solves issue whether receiver receives correct message from the correct communicator)
- st — status of the receive command; its info includes MPI_SOURCE, MPI_TAG, MPI_ERROR, and message length

A Simple Send/Receive Program

```
#include <mpi.h>

int main(int argc, char **argv) {
  int i, rank, size, dest, to, src, from;
  int count, tag, st_count, st_src, st_tag;
  double data[100];
  MPI_Status st;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Process %d of %d is alive\n", rank, size);
  src = 0; dest = size - 1;

  if (rank == src) {
    to = dest;
    tag = 2001;
    count = 100;
    for (i = 0; i < 100; i++)
      data[i] = i;
    MPI_Send(data, count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);
  }
```

A Simple Send/Receive Program (cont.)

```
  else if (rank == dest) {
    from = MPI_ANY_SOURCE;
    tag = MPI_ANY_TAG;
    count = 100;
    MPI_Recv(data, count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &st);
    MPI_Get_count(&st, MPI_DOUBLE, &st_count);
    st_src = status.MPI_SOURCE;
    st_tag = status.MPI_TAG;
    printf("Status info: source = %d, tag = %d, count = %d\n",
           st_src, st_tag, st_count);

    printf(" %d received: ", rank);
    for (i = 0; i < st_count; i++)
      printf("%lf ", data[i]);
    printf("\n");
  }
  MPI_Finalize();
  return 0;
}
```


Non-Blocking Send and Receive

I stands for immediate return.

```
int MPI_ISend(void *buf, int count, MPI_Datatype type, int dest,
              int tag, MPI_COMM comm, MPI_Request *req)
```

```
int MPI_IRecv(void *buf, int count, MPI_Datatype type, int src,
              int tag, MPI_COMM comm, MPI_Request *req)
```

Both routines create and allocate a *request object* and return a pointer to it in the `req` variable.

This object is useful for future testing and waiting for the finish of the non-blocking operation:

```
int MPI_Test(MPI_Request *reg, int *flag, MPI_Status *st)
int MPI_Wait(MPI_Request *reg, MPI_Status *st)
```

- If the value of `flag` is `true`, the operation has completed.

Combined Send/Receive

```
int MPI_Sendrecv(void *sbuf, int scnt, MPI_Datatype stype, int dest,
                 int stag, void *rbuf, int rcnt, MPI_Type rtype,
                 int src, int rtag, MPI_COMM comm, MPI_Status *st)
```

The routine performs both a send and a receive. Advantages over pair of send and receive:

- avoids deadlock
- does not require data buffering
- Safe?

Other Send Routines

- Synchronous Sends — Won't complete until the corresponding receive has been posted:

```
int MPI_Ssend(...)
int MPI_Issend(...)
```

- Ready Sends — Assume the corresponding receive has been posted; otherwise they are errorous:

```
int MPI_Rsend(...)
int MPI_Irsend(...)
```

- Buffered Sends — If the corresponding receive has not been posted, the system must buffer the data and return; it's the user's responsibility to allocate the buffer:

```
int MPI_Bsend(...)
int MPI_Ibsend(...)
```

Collective Communications (Sync, Must happen synchronously till the end)

MPI Collective communications are coordinated among a group of processes, as specified by communicator.

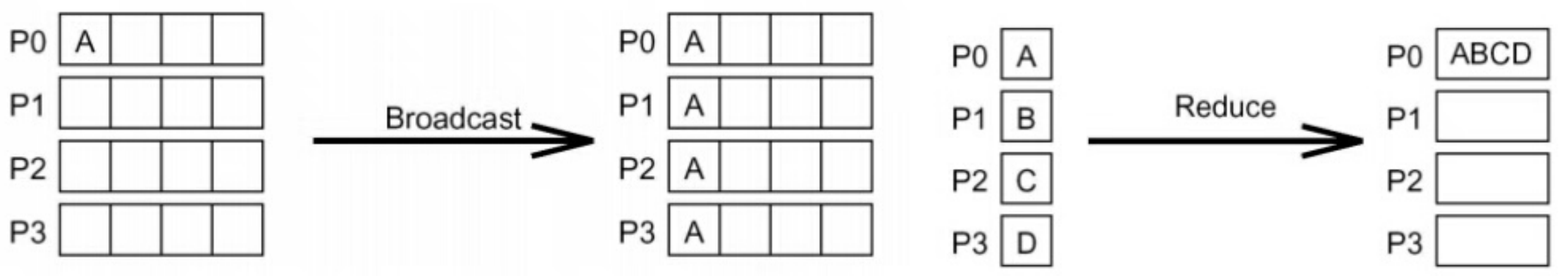
- Message tags are not used
- All collective operations are blocking
- All processes in the communicator group must call the collective operation

Collective Communicators are implemented with

Categories:

- One-to-All Broadcast
- All-to-One Reduction
- Scatter and Gather
- Prefix Scan
- All-to-All Broadcast & Reduction

Basic Broadcast and Reduce




```
int MPI_Bcast(void* buf, int cnt, MPI_Datatype type, int root,
              MPI_Comm comm)
```

→ source of broadcast

```
int MPI_Reduce(void* sbuf, void* rbuf, int cnt, MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
```

→ receiver of the reduce operation

- The reduce routine take can take the following built-in collective operators:

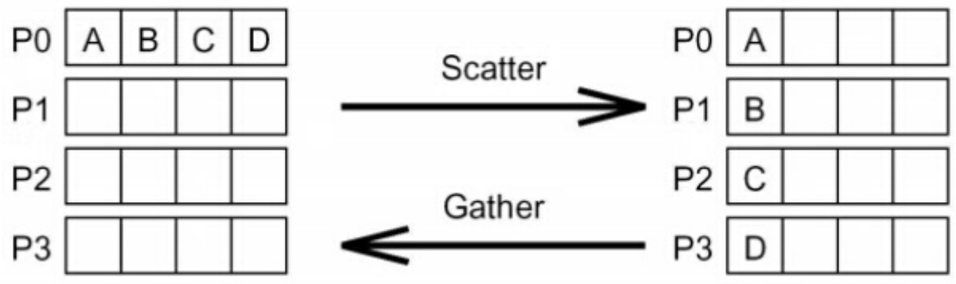
```
MPI_SUM   MPI_MAX   MPI_LAND   MPI_BAND   MPI_LXOR   MPI_MAXLOC
MPI_PROD  MPI_MIN   MPI_LOR    MPI_BOR    MPI_BXOR   MPI_MINLOC
```

- It can also take user-defined reduction functions.

Parallel Programming is complicated!!!

Scatter and Gather → Reverse of scatter.

→ Auto partition among participating processes




```
int MPI_Scatter(void* sbuf, int scnt, MPI_Datatype stype,
                void *rbuf, int rcnt, MPI_Datatype rtype,
                int source, MPI_Comm comm)
```

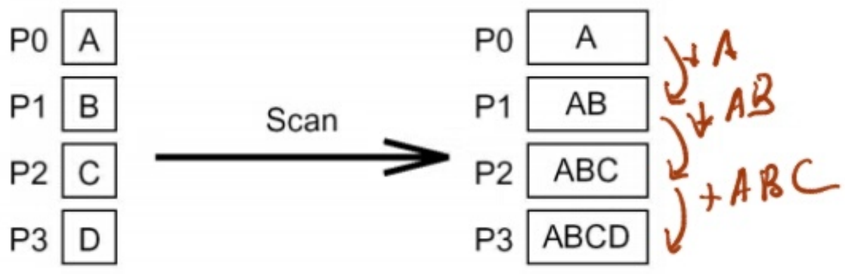
```
int MPI_Gather(void* sbuf, int scnt, MPI_Datatype stype,
               void *rbuf, int rcnt, MPI_Datatype rtype,
               int target, MPI_Comm comm)
```

Don't have much control over it.

All processes must execute the call!

Prefix Scan? (What the heck?)

Partial results are computed and stored in participating nodes.



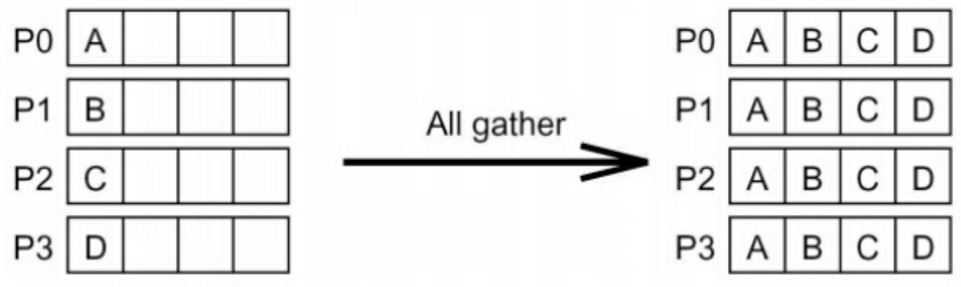

```
int MPI_Scan(void* sbuf, void *rbuf, int cnt, MPI_Datatype stype,
             MPI_Op op, MPI_Comm comm)
```

Circled churn like dpsp to avoid exponential explosion
Expensive (should be avoided, at least linear performance increase)
→ When Reduce → all get copy of the result
→ When broadcast → all broadcast to all

All-Version and V-Version Routines

↳ allows contribution of different sizes

- All-versions deliver results to all participating processes.



`MPI_ALLGATHER` `MPI_ALLREDUCE` `MPI_ALLTOALL`

- V-versions allow the chunks to have different sizes. (Complicated)

  `MPI_ALLGATHERV` `MPI_GATHERV` `MPI_ALLTOALLV`

Example: Collective Communication

```c
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, myn, i, N;
  double *vector, *myvec, sum, mysum, total;

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

  /* In the root process read the vector length, initialize
     the vector and determine the sub-vector sizes */
  if (rank == 0) {
    printf("Enter the vector length : ");
    scanf("%d", &N);
    vector = (double *)malloc(sizeof(double) * N);
    for (i = 0, sum = 0; i < N; i++)
      vector[i] = 1.0;
    myn = N / size;
  }

  MPI_Bcast(&myn, 1, MPI_INT, 0, MPI_COMM_WORLD);
```


MPI is a collective routine and s
collective routines should not be called within conditional statements.

Example: Collective Communication (cont.)

```
  myvec = (double *)malloc(sizeof(double)*myn);
  MPI_Scatter(vector, myn, MPI_DOUBLE, myvec, myn, MPI_DOUBLE,
              0, MPI_COMM_WORLD);

  /* Find the local sum, then the global sum of the vectors */
  for (i = 0, mysum = 0; i < myn; i++)
    mysum += myvec[i];
  MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  /* Multiply the local part of the vector by the global sum */
  for (i = 0; i < myn; i++)
    myvec[i] *= total;
  MPI_Gather(myvec, myn, MPI_DOUBLE, vector, myn, MPI_DOUBLE,
             0, MPI_COMM_WORLD);

  if (rank == 0)
    for (i = 0; i < N; i++)
      printf("[%d] %f\n", rank, vector[i]);
  MPI_Finalize();
  return 0;
}
```

every node will receive sum.

Advanced features further on (for information only)!

One-Sided Communications

(*A.k.a.* Remote memory access or RMA.) Remote memory read/write

- Allow one process to specify all communication parameters, both for the sending side and for the receiving side.
- De-couple communication of data and syncrhonization between sender and receiver.
- The semantics is *very* complicated.

Sample Routines:

- `MPI_Put` — remote write
- `MPI_Get` — remote read
- `MPI_Accumulate` — remote update
- `MPI_Compare_and_swap` — remote atomic swap

Seems convinient but hard to keep memory consistency.

Two main constraints

One-Sided Communications (cont.)

1 ▶ One-sided communications can only access data in specific memory regions called "*windows*", which have to be explicitly created.

```
MPI_Win_create(...) // windows where RMA is allowed!
```

2. ▶ One-sided communications can only occur during specific temporal intervals called "*epochs*", which are bracketed with synchronization calls.

```
MPI_Win_lock(...)
... // RMA operation here
MPI_Win_unlock(...)
```

RMA only allowed in some temporal interval. This defines the interval.

Example: One-Sided Communication

Process A:

```
int n;
MPI_WIN nwin;

MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
...
n = 1000; /* local update */
MPI_Barrier(MPI_COMM_WORLD);
...
```

flag to indicate that window is ready.

Process B:

```
int n;
MPI_WIN nwin;

MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
...
MPI_Barrier(MPI_COMM_WORLD); //wait for the flag to go up
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, nwin);
MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin); /* remote read */
MPI_Win_unlock(MPI_LOCK_EXCLUSIVE, 0, 0, nwin);
```

Read can happen concurently. All processes assumes that there is no other readers, i.e. other reads dont advance the pointer.

File I/O

How to povide concurent access?

Each process will open the same file

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
                  MPI_Info info, MPI_File *fh)
```

```
int MPI_File_close(MPI_File *fh)
```

- Both routines are collective routines:
  – Input parameters (except `info`) must have the same values;
  – `MPI_File_close` implies a `MPi_File_sync`.
- However, it's possible to open a file to just one process — by using `MPI_COMM_SELF` as `comm`'s value.
- File access modes include the usual choices, *e.g.*, *read-only*, *read-write*, *append*, etc. MPI_MODE_RDONLY
- File Info is for advanced uses (i.e. passing optimization hints). For normal cases, just use `MPI_INFO_NULL`.

Example: Reading From a File

called by every process

calle before reading/writing.

```
int main(int argc, char *argv[])
{
  int size, rank, buf[2];
  MPI_File fh;
  MPI_Status st;

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

  MPI_File_open(MPI_COMM_WORLD, argv[1], MPI_MODE_RDONLY,
                MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL); // specify
  MPI_File_read(fh, buf, 2, MPI_INT, &st);
  MPI_File_close(&fh);

  printf("rank=%d: buf=[%d,%d]\n", rank, buf[0], buf[1]);
  MPI_Finalize();
  return 0;
}
```

sections of the file (partition)

Example: Writing to a File

```c
int main(int argc, char *argv[])
{
  int cnt=4, buf[4], i, rank;
  char fname[10];
  MPI_File fh;
  MPI_Status st;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for (i=0; i<cnt; i++) buf[i] = rank*100 + i;
  sprintf(fname, "%s.%d", "test", rank);

  MPI_File_open(MPI_COMM_SELF, fname, MPI_MODE_CREATE|MPI_MODE_RDWR,
                MPI_INFO_NULL, &fh);
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_write(fh, buf, cnt, MPI_INT, &st);
  MPI_File_close(&fh);

  MPI_Finalize();
  return 0;
}
```


File View

```c
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype ftype, char *datarep, MPI_Info info)
```

This routine sets individual process's view of the data in the file.

Key →

- `disp` — displacement. Specifies where within the file to start access. Useful for reading different sections to different processes.
- `etype` — elementary datatype.
- `ftype` — file type. For advanced uses. Specifies gaps between desirable data items. For simple cases, just use the same value as `etype`.
- `datarep` — data representation. Normally just use `"native"`.

Open MPI

An open-source implementation of MPI.

- Comprehensive — support many platforms (including GPUs)
- Flexible — lots of user-level configuration/tuning controls
- High-Performance — used by many TOP500 supercomputers

(The name 'Open MPI' can be shortend to OMPI, but *not* to OpenMPI.)

The steps to use Open MPI:

- Creating a hostfile
- Compiling a program
- Running a program

Optionally, one can also create an appfile.

Open MPI: Creating a Hostfile

A hostfile contains a list of computer names.

Example:

```
linux> cat myhosts
# a line starting with # is a comment
african
chatham
chinstrap
...
```

- The MPI program must be accessible by the same pathname on all host computers.
- The environment variable `OMPI_MCA_orte_default_hostfile` can be set to point to a default hostfile.
- It's a good idea to create multiple host files for different number of hosts: *e.g.* `mpihost2`, `mpihost4`, `mpihost8`, etc.

Open MPI: Compiling a Program

The command: **mpicc**

Example:

```
linux> mpicc -o ring ring.c
```

- mpicc is just a gcc wrapper:

```
linux> mpicc --showme
gcc -I/usr/lib/openmpi/include -I/usr/lib/openmpi/include/openmpi
-pthread -L/usr/lib/openmpi/lib -lmpi -lopen-rte -lopen-pal -ldl
-Wl,--export-dynamic -lnsl -lutil -lm -ldl
```


Open MPI: Running a Program

The command:

mpirun -host <hostnames> -n <#process> <program>

mpirun -hostfile <hostfile> -n <#process> <program>

Example:

```
linux> mpirun -n 4 ring
linux> mpirun -host african -n 4 ring
linux> mpirun -host african,chatham -n 4 ring
linux> mpirun -hostfile host2 -n 4 ring
linux> mpirun -hostfile host4 -n 4 ring
```

- If there is no host or hostfile specified, the program will run on the console computer.
- The number of processes do not need to match the number of hosts.
- If a default hostfile exists, the `-hostfile` switch can be omitted.

Open MPI: Running a Program (cont.)

Open MPI also supports master/slave programs:

Example:

```
linux> mpirun -hostfile myhosts -n 1 master -n 4 slave
```

This command will run one copy of 'master' and four copies of 'slave'.

Creating an Appfile

mpirun command-line parameters can be saved in an appfile; and a program can be executed from an appfile.

Example:

```
linux> cat myapp
# execute 4 copies of 'ring' on 2 hosts
-host african,chatham -n 4 ring
linux> mpirun -app myapp
```