MPI — Point to Point communications

Salvatore Filippone

salvatore.filippone@uniroma2.it

Getting started

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

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

    MPI_Finalize();
  }
```

- Everything should happen between `MPI_Init` and `MPI_Finalize`;
- Compile with `mpicc`

Point-to-point communication

Essential questions:

- Who are you?
- With whom do you have to cooperate?

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

- The set of processes that you are using is established *outside* of the program;
- The value of `my_rank` is between 0 and $p-1$ or `MPI_UNDEFINED`.
- The value of `p` is a positive number;

Point-to-point communication

- Define what data should be sent and received;
- Define how much data should be sent and received;

```
MPI_Send(message,strlen(message)+1,MPI_CHAR,
         dest,tag,MPI_COMM_WORLD);

MPI_Recv(message,100,MPI_CHAR,
         source,tag,MPI_COMM_WORLD,&status);
```

Basic data types

MPI type	C type
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Message Contents

Basic data types

MPI type	C type
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Type matching

- The type in a program matches the type in the communication if the datatype name corresponds to the basic type of the program variable.
- The types of a send and receive match if both operations use identical names. That is, `MPI_INT` matches `MPI_INT`, `MPI_REAL` matches `MPI_REAL`, and so on.
- Exception: `MPI_PACKED` can match any other type.

Message Contents

Why the type system?

Heterogeneous environments

If the parallel program is running in a heterogeneous environment, the type system guarantees that the data is converted as appropriate

If you want to send the data *without* any conversion (or interpretation) use `MPI_BYTE`.

Message Envelope

How to distinguish between similar messages?

Envelope of a message

- Rank of the receiver;
- Rank of the sender;
- Tag;
- Communicator.

Message Envelope

How to distinguish between similar messages?

Envelope of a message

- Rank of the receiver;
- Rank of the sender;
- Tag;
- Communicator.

Implementation guarantee

Any two messages with the same envelope will always be delivered in the correct order

The programmer can count on this, the MPI implementation *must* work this way.

P2P Communication Semantics

Order If two messages from the same sender match a receive, they are received in the order they were sent;

Progress If a matching send/receive pair is initiated, at least one of the operations will complete;

Fairness There is no guarantee that a message will ever be received[1];

Resource limitations Buffer space will be consumed, and this may cause errors.

The full description of these rules is much more complicated.

Note: in a multithreaded environment `SEND` operations from different threads may execute in any order.

[1] could be overtaken by messages from other sources

P2P Communication Semantics

All communications we have seen are blocking; but what does it mean?

All communications we have seen are blocking; but what does it mean?

Buffering for SEND operations

When does an `MPI_Send` call terminate?

When the data in the SEND buffer has been stored away, so that the calling program is free to modify it

Two alternatives:

- The data has been copied to a buffer local to the sending process; Locally Blocking (potentially much faster);
- The data has been copied to the matching receive buffer (i.e. it's already at its destination); Globally Blocking;

P2P Communication Semantics

All communications we have seen are blocking; but what does it mean?

Buffering for SEND operations

When does an `MPI_Send` call terminate?

When the data in the SEND buffer has been stored away, so that the calling program is free to modify it

Two alternatives:

- The data has been copied to a buffer local to the sending process; Locally Blocking (potentially much faster);
- The data has been copied to the matching receive buffer (i.e. it's already at its destination); Globally Blocking;

Locally or globally blocking?

It is up to the implementation to decide whether to buffer locally (it can be expensive!)

Hence, a send *may or may not* need a matching receive to complete.

P2P Communication Semantics

Safe

```
IF (rank == 0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 1, tag, comm, status, ierr)
ELSE IF (rank == 1) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 0, tag, comm, ierr)
END IF
```

P2P Communication Semantics

Wrong

```
IF (rank == 0) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 1, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 1, tag, comm, ierr)
ELSE IF (rank == 1) THEN
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 0, tag, comm, status, ierr)
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 0, tag, comm, ierr)
END IF
```

P2P Communication Semantics

Potentially Unsafe

```
IF (rank == 0) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 1, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 1, tag, comm, status, ierr)
ELSE IF (rank == 1) THEN
  CALL MPI_SEND(sendbuf, count, MPI_REAL,&
       & 0, tag, comm, ierr)
  CALL MPI_RECV(recvbuf, count, MPI_REAL,&
       & 0, tag, comm, status, ierr)
END IF
```

P2P Communication modes

Standard mode: What we have seen so far, the implementation decides what to do;

Buffered mode: The user provides a buffer for the outgoing message with `MPI_Buffer_attach`; the call to `MPI_Bsend` completes immediately (locally blocking);

Synchronous mode: A call to `MPI_Ssend` can start at any time, but can complete only when a matching receive has been posted and has started (globally blocking);

Ready mode: A call to `MPI_Rsend` can be started *only* if a matching receive has already beeen posted, and is erroneous otherwise (outcome is undefined).

Non-blocking Point-to-point communication

Safe

```
if (rank==0)  {
  MPI_Irecv(recvbuf, count, MPI_REAL,
            1, tag, comm, &request);
  MPI_Send(sendbuf, count, MPI_REAL,
           1, tag, comm);
  MPI_Wait(&request,&status);
 } else if (rank==1) {
  MPI_Irecv(recvbuf, count, MPI_REAL,
            0, tag, comm, &request);
  MPI_Send(sendbuf, count, MPI_REAL,
           0, tag, comm);
  MPI_Wait(&request,&status);
 }
```

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;
- The `request` object can be used in a subsequent call for completion:

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;
- The `request` object can be used in a subsequent call for completion:
    - `MPI_Wait`

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;
- The `request` object can be used in a subsequent call for completion:
  - `MPI_Wait`
  - `MPI_Iprobe`

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;
- The `request` object can be used in a subsequent call for completion:
  - `MPI_Wait`
  - `MPI_Iprobe`
- Wait for multiple calls: `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`

Non-blocking Point-to-point communication

- Each non-blocking call returns immediately;
- The user *must not* modify the send buffer before checking for completion;
- The user *should not* use the contents of the receive buffer before checking for completion;
- Each call uses an opaque `MPI_Request` object that the application can use to keep track of the communication;
- The `request` object can be used in a subsequent call for completion:
  - `MPI_Wait`
  - `MPI_Iprobe`
- Wait for multiple calls: `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`
- Details of the received message: the request contains fields for `TAG`, `SOURCE` and `ERROR` that can be accessed directly; you can also query with `MPI_Get_count`.

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Thus achieving overlap between communication and computations.

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Thus achieving overlap between communication and computations.

Overlap

Question: Can you *actually* get overlap between communication and computations?

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Thus achieving overlap between communication and computations.

Overlap

Question: Can you *actually* get overlap between communication and computations?

Answer: Maybe

Non-blocking Point-to-point communication

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Thus achieving overlap between communication and computations.

Overlap

Question: Can you *actually* get overlap between communication and computations?

Answer: Maybe

They are appealing because you can:

- Do a call to `MPI_Isend` or `MPI_Irecv`;
- Go perform other tasks (preserving the buffer status/contents);
- Come back and check for completion

Thus achieving overlap between communication and computations.

Overlap

Question Can you *actually* get overlap between communication and computations?

Answer Maybe

It depends on the implementation; the definition of `progress` in the standard is carefully crafted to allow implementations that do *not* provide overlap.

Derived Datatypes

There is a mechanism to define composite data types, combining structures and/or sections of arrays:

- `MPI_Type_contiguous`
- `MPI_Type_create_struct`
- `MPI_Type_vector`
- `MPI_Type_indexed`
- `MPI_Type_create_struct`
- `MPI_Type_commit`, `MPI_Type_dup`, `MPI_Type_free`

Why use them?

- You can delegate to the MPI implementation the collection of the data items for packing;
- The MPI implementation may choose to skip the packing into an intermediate buffer, with a performance advantage.

If you need, you can invoke explicitly `MPI_Pack`, `MPI_Unpack`.

Definition

The *extent* of a datatype is the number of bytes in memory

Derived Datatypes Example

```
 MPI_Type_contiguous(2, MPI_REAL, type2, ...);
 MPI_Type_contiguous(4, MPI_REAL, type4, ...);
 MPI_Type_contiguous(2, type2, type22, ...);
 MPI_Type_commit(type2);
 MPI_Type_commit(type4);
 MPI_Type_commit(type22);
...
 MPI_Send(a, 4, MPI_REAL, ...);
 MPI_Send(a, 2, type2, ...);
 MPI_Send(a, 1, type22, ...);
 MPI_Send(a, 1, type4, ...);
...
 MPI_Recv(a, 4, MPI_REAL, ...);
 MPI_Recv(a, 2, type2, ...);
 MPI_Recv(a, 1, type22, ...);
 MPI_Recv(a, 1, type4, ...);


 MPI_Type_free(type2);
 MPI_Type_free(type4);
 MPI_Type_free(type22);
```

Derived Datatypes Example

```
struct Partstruct
{
  int
  type; /* particle type */
  double d[6];
  /* particle coordinates */
  char
  b[7];
  /* some additional information */
};
struct Partstruct
particle[1000];
int
MPI_Comm
i, dest, tag;
comm;
/* build datatype describing structure */
MPI_Datatype Particlestruct, Particletype;
MPI_Datatype  type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int  blocklen[3] = {1, 6, 7};
MPI_Aint  disp[3];
MPI_Aint base, lb, sizeofentry;
```

Derived Datatypes Example

```
/* compute displacements of structure components */
MPI_Get_address(particle, disp);
MPI_Get_address(particle[0].d, disp+1);
MPI_Get_address(particle[0].b, disp+2);
base = disp[0];
for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);
MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
/* If compiler does padding in mysterious ways,
   the following may be safer */
/* compute extent of the structure */
MPI_Get_address(particle+1, &sizeofentry);
sizeofentry = MPI_Aint_diff(sizeofentry, base);
/* build datatype describing structure */
MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);
/* 4.1:
   send the entire array */
MPI_Type_commit(&Particletype);
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
/* 4.2:
   send only the entries of type zero particles,
   preceded by the number of such entries */
MPI_Datatype Zparticles;
MPI_Datatype Ztype;
```

Derived Datatypes Example

```
/* datatype describing all particles
   with type zero (needs to be recomputed
   if types change) */
int zdisp[1000];
int zblock[1000], j, k;
int zzblock[2] = {1,1};
MPI_Aint zzdisp[2];
MPI_Datatype zztype[2];
/* compute displacements of type zero particles */
j = 0;
for (i=0; i < 1000; i++)
  if (particle[i].type == 0)
    {
      zdisp[j] = i;
      zblock[j] = 1;
      j++;
    }
```

Derived Datatypes Example

```
/* create datatype for type zero particles */
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
/* prepend particle count */
MPI_Get_address(&j, zzdisp);
MPI_Get_address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
MPI_Type_commit(&Ztype);
MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
/* A probably more efficient way of defining Zparticles */
/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].type == 0)
    {
      for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);
      zdisp[j] = i;
      zblock[j] = k-i;
      j++;
      i = k;
    }
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
```

Derived Datatypes Example

```
/* 4.3:
   send the first two coordinates of all entries */
MPI_Datatype Allpairs;
/* datatype for all pairs of coordinates */
MPI_Type_get_extent(Particletype, &lb, &sizeofentry);
/* sizeofentry can also be computed by subtracting the address
   of particle[0] from the address of particle[1] */
MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit(&Allpairs);
MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
/* an alternative solution to 4.3 */
MPI_Datatype Twodouble;
MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
MPI_Datatype Onepair;
/* datatype for one pair of coordinates, with
   the extent of one particle entry */
MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
MPI_Type_commit(&Onepair);
MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
```