MPI

Jiarui Guo @ ZJUSCT

July 7, 2024

Outline

Introduction

History

- Before 1990's
  Many libraries.
  Writing code was a difficult and tedious task.

  Models commonly adopted: Message Passing Model
  An application passes messages among processes in order to perform a task.
  e.g. job assignment, results of sub-problems...

- Supercomputing 1992 conference
  A standard interface was defined.
- 1994
  MPI-1
- 2023
  MPI-4.1 by MPI Forum.

What is MPI

MPI, a Message Passaing Interface.

There exists many implementations:

- OpenMPI
- Intel-MPI
- HMPI (Hyper-MPI)
- MPICH
- ......

Installation

- OpenMPI
  Lab0
- Intel-MPI
  Intel-oneAPI(Click Me)
- HMPI
  Huawei (Click Me)

Hello World!

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    printf("Hello␣world␣from␣processor␣%s,␣rank␣%d␣out␣of␣%d␣processors\n",
    processor_name, world_rank, world_size);
    MPI_Finalize();
    return 0;
}
```

Basic Concepts

Communicator

A communicator defines a group of processes that have the ability to communicate with one another.

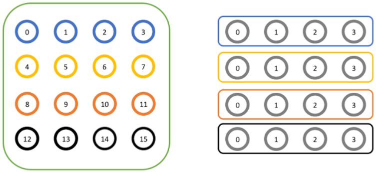
Each process has a unique rank.

Communicator(cont.)

- MPI_COMM_WORLD

- MPI_COMM_SPLIT(comm, color, key, new_comm)
  - comm: The communicator that will be used as the basis for the new communicators.
  - color: Which new communicator each processes will belong.
  - key: The ordering (rank) within each new communicator.
  - new_comm: [OUT]

Split a Large Communicator Into Smaller Communicators



Blocking vs Non-blocking

Blocking

It does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer.

The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Non-blocking

A nonblocking call initiates the operation, but does not complete it.

They will return almost immediately.

Messages are non-overtaking

Order is preserved.(Only under single thread)

If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.

If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

Fairness

MPI makes no guarantee of fairness in the handling of communication.

There may be starvation.

Example

Rank1 $\rightarrow^{send}$ Rank0

Rank2 $\rightarrow^{send}$ Rank0

Rank0 $\leftarrow^{receive}$ from any source.

Point-to-Point Communication

Blocking Send and Receive

```
int MPI_Send(const void* buffer,
             int count,
             MPI_Datatype datatype,
             int recipient,
             int tag,
             MPI_Comm communicator);
```

Parameters:

- **buffer** The buffer to send.
- **count** The number of elements to send.
- **datatype** The type of one buffer element.
- **recipient** The rank of the recipient MPI process.
- **tag** The tag to assign to the message.
- **communicator** The communicator in which the standard send takes place.

Blocking Send and Receive

```c
int MPI_Recv(void* buffer,
        int count,
        MPI_Datatype datatype,
        int sender,
        int tag,
        MPI_Comm communicator,
        MPI_Status* status);
```

Parameters:

- **buffer** The buffer to receive.
- **count** The number of elements to receive.
- **datatype** The type of one buffer element.
- **sender** The rank of the sender MPI process.
- **tag** The tag to assign to the message.
- **communicator** The communicator in which the standard receive takes place.
- **status** The variable in which store the status of the receive operation. Pass MPI_STATUS_IGNORE if unused.

MPI_Status

MPI_Status represents the status of a reception operation.

At least 3 attributes:

- MPI_SOURCE
- MPI_TAG
- MPI_ERROR

There may be additional attributes that are implementation-specific.

Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them.

- source
- destination
- tag
- communicator

Communication Mode

- **Buffer Mode**
  Can be started whether or not a matching receive was posted. Completion does not depend on the occurrence of a matching receive.
- **Synchronous Mode**
  Can be started whether or not a matching receive was posted.
  The send will be completed successfully only if a matching receive is posted.
- **Ready Mode**
  May be started only if the matching receive is already posted.
- **Standard Mode**
  Depends.

Communication Mode

Communication mode	**Start time**	**Completion time**
Buffer mode	Immediately	Message has gone to buffer
Synchronous mode	Immediately	Matching receive has posted
Ready mode	Matching receive has posted	When the send buffer can be reused
Standard mode	Depends	Depends

Example

```
// n = 2
MPI_Comm_rank(comm, &my_rank);
MPI_Ssend(sendbuf, count, MPI_INT, my_rank ^ 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_INT, my_rank ^ 1, tag, comm, &status);
```

Example

```
// n = 2
MPI_Comm_rank(comm, &my_rank);
MPI_Ssend(sendbuf, count, MPI_INT, my_rank ^ 1, tag, comm);
MPI_Recv(recvbuf, count, MPI_INT, my_rank ^ 1, tag, comm, &status);
```

Deadlock! Any solutions?

Example

```
// n = 2
MPI_Comm_rank(comm, &my_rank);
if (my_rank == 0) {
    MPI_Ssend(sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (my_rank == 1) {
    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
    MPI_Ssend(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Any other solutions?

Blocking Send and Receive

```
int MPI_Sendrecv(const void* buffer_send,
                 int count_send,
                 MPI_Datatype datatype_send,
                 int recipient,
                 int tag_send,
                 void* buffer_recv,
                 int count_recv,
                 MPI_Datatype datatype_recv,
                 int sender,
                 int tag_recv,
                 MPI_Comm communicator,
                 MPI_Status* status);
```

Notice

The buffers used for send and receive must be different.

Any other solutions?

Non-Blocking Send and Receive

Recall:

A nonblocking call initiates the operation, but does not complete it.

They will return almost immediately.

```
int MPI_Isend(const void* buffer,
        int count,
        MPI_Datatype datatype,
        int recipient,
        int tag,
        MPI_Comm communicator,
        MPI_Request* request);
```

- **MPI_Test**
  MPI_TEST(request, flag, status)
  Checks if a non-blocking operation is complete at a given time.
  flag=true if completes.

- **MPI_Wait**
  MPI_WAIT(request, status)
  Waits for a non-blocking operation to complete. That is, unlike MPI_Test, MPI_Wait will block until the underlying non-blocking operation completes.

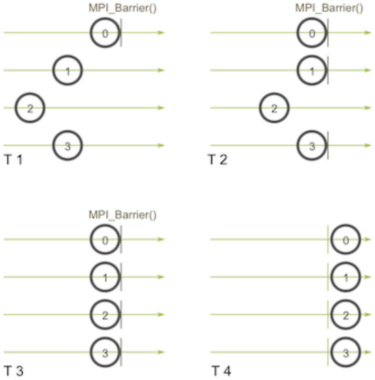
Non-Blocking Send and Receive(Deadlock revisit)

```
MPI_Request req;
MPI_Isend(sendbuf, 0x100, MPI_INT, my_rank^1, 0, MPI_COMM_WORLD, &req);
MPI_Recv(recvbuf, 0x100, MPI_INT, my_rank^1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Collective Communication

Synchronization

- **MPI_Barrier**
  MPI_Barrier(COMM)
  Blocks all MPI processes in the given communicator until they all call this routine.

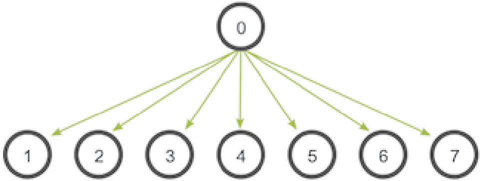


MPI_Barrier

BroadCast(One to All)

```
int  MPI_Bcast(void* buffer,
          int  count,
          MPI_Datatype datatype,
          int  emitter_rank ,
          MPI_Comm communicator);
```

- **emitter_rank** The rank of the MPI process that broadcasts the data, all other processes receive the data broadcasted.

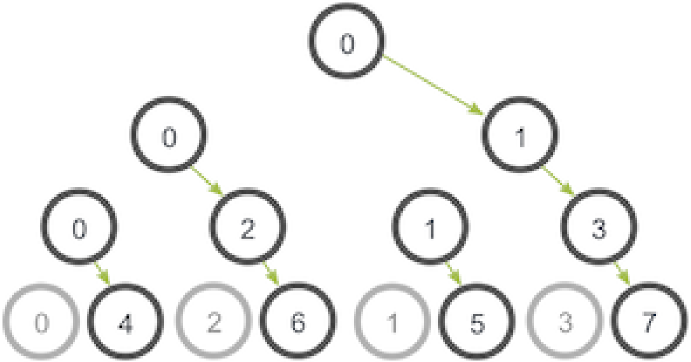


Bcast

Why not Send and Receive?

```
double start = MPI_Wtime();
if (my_rank == 0){
    for (int i=1; i<=31; i++)
        MPI_Send(sendbuf, 0x10000, MPI_INT, i, 0, MPI_COMM_WORLD);
}else{
    MPI_Recv(recvbuf, 0x10000, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
double end = MPI_Wtime();
if (my_rank == 0) printf("[Send-Recv] Finished in %f seconds\n", my_rank, end-start);
start = MPI_Wtime();
MPI_Bcast(&sendbuf, 0x10000, MPI_INT, 0, MPI_COMM_WORLD);
end = MPI_Wtime();
if (my_rank == 0) printf("[Bcast] Finished in %f seconds\n", my_rank, end-start);
```

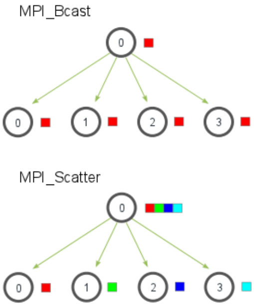
BroadCast(Tree based algorithm)



Scatter(One to All)

```c
int MPI_Scatter(const void* buffer_send ,
      int count_send,
      MPI_Datatype datatype_send,
      void* buffer_recv ,
      int count_recv,
      MPI_Datatype datatype_recv,
      int root,
      MPI_Comm communicator);
```

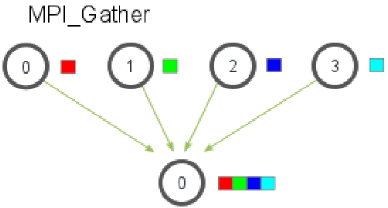
- **count_send** The number of elements to send to each process, not the total number of elements in the send buffer. For non-root processes, the send parameters like this one are ignored.
- **count_receive** The number of elements in the receive buffer.



Scatter

Gather(All to One)

```
int MPI_Gather(const void* buffer_send,
    int count_send,
    MPI_Datatype datatype_send,
    void* buffer_recv ,
    int count_recv ,
    MPI_Datatype datatype_recv,
    int root ,
    MPI_Comm communicator);
```



MPI_Gather

Example

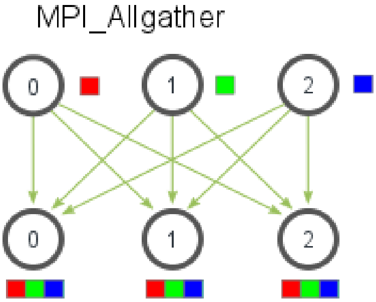
Compute average

```
MPI_Scatter(buffer, 0x1000000/4, MPI_DOUBLE, local_buffer, 0x1000000/4, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
double local_avg = 0;
for(int i=0; i<0x1000000/4; i++){
    local_avg += local_buffer[i];
}
local_avg /= 0x1000000/4;
double avgs[4];
MPI_Gather(&local_avg, 1, MPI_DOUBLE, avgs, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Allgather(All to All)

```
int MPI_Allgather(const void* buffer_send,
        int count_send,
        MPI_Datatype datatype_send,
        void* buffer_recv,
        int count_recv,
        MPI_Datatype datatype_recv,
        MPI_Comm communicator);
```

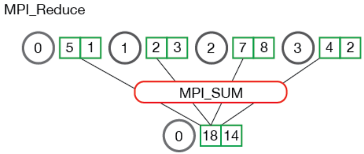
Actually MPI_Gather + MPI_Bcast.



MPI_Allgather

Reduce

```c
int MPI_Reduce(const void* send_buffer,
    void*  receive_buffer ,
    int count,
    MPI_Datatype datatype,
    MPI_Op operation,
    int root,
    MPI_Comm communicator);
```



Reduce

Reduce

Example

Compute average revisit

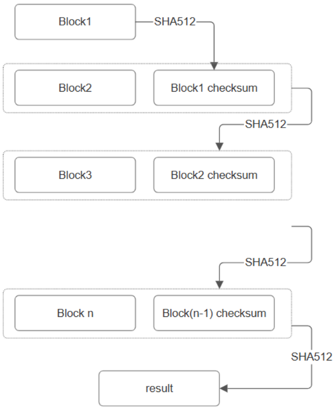
```
MPI_Scatter(buffer, 0x1000000/4, MPI_DOUBLE, local_buffer, 0x1000000/4, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
double local_avg = 0;
for(int i=0; i<0x1000000/4; i++){
      local_avg += local_buffer[i];
}
local_avg /= 0x1000000/4;
double global_avg;
MPI_Reduce(&local_avg, &global_avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Example

Task

Implement a data validation algorithm using SHA512.
Algorithm procedure:

1. Tile the input file into blocks of 1MB. (If the last block is smaller than 1MB, pad it with zeros.)
2. For the i^{th} block, concatenate it with the validation sum SHA512 of $(i-1)^{th}$ block and calculate validation sum of SHA512.
3. The validation sum of the last block is considered as the validation sum of the entire file.



Baseline Code

```
int num_block = (len + BLOCK_SIZE - 1) / BLOCK_SIZE;
uint8_t prev_md[SHA512_DIGEST_LENGTH];

EVP_MD_CTX *ctx = EVP_MD_CTX_new();
EVP_MD *sha512 = EVP_MD_fetch(nullptr, "SHA512", nullptr);

SHA512(nullptr, 0, prev_md);
```

```
for (int i = 0; i < num_block; i++) {
    uint8_t buffer[BLOCK_SIZE]{};
    EVP_DigestInit_ex(ctx, sha512, nullptr);
    std::memcpy(buffer, data + i * BLOCK_SIZE,
                std::min(BLOCK_SIZE, len - i * BLOCK_SIZE));
    EVP_DigestUpdate(ctx, buffer, BLOCK_SIZE);
    EVP_DigestUpdate(ctx, prev_md, SHA512_DIGEST_LENGTH);

    unsigned int len = 0;
    EVP_DigestFinal_ex(ctx, prev_md, &len);
```

Notice

EVP_DigestUpdate(a); EVP_DigestUpdate(b);

Equivalent to

EVP_DigestUpdate(concate(a,b)) !

Computation is dependent on the result of the previous one.

How to exploit MPI?

Analysis

Computation is dependent on the result of the previous one.

How to exploit MPI?

Answer:

File **I/O** accounts! We can **overlap** I/O operations with computation.

MPI Code

Non-Blocking receives the previous block's checksum.

```
if (i != 0){
  MPI_Irecv((void *)prev_md,
            SHA512_DIGEST_LENGTH,
            MPI_UINT8_T,
            sender,
            0,
            MPI_COMM_WORLD,
            &request);
}
```

Meanwhile... File I/O and Digest

```
istrm.seekg(i * BLOCK_SIZE);
istrm.read( reinterpret_cast <char *>(data + i * BLOCK_SIZE), std::
        min(BLOCK_SIZE*local_size, file_size - i * BLOCK_SIZE));

for (int j=i; j<upper_bound; j++){
  uint8_t buffer2[BLOCK_SIZE]{};
  EVP_DigestInit_ex(ctx[j-i], sha512, nullptr);
  std::memcpy(buffer2, data + j * BLOCK_SIZE,
              std::min(BLOCK_SIZE, len - j * BLOCK_SIZE));
  EVP_DigestUpdate(ctx[j-i], buffer2, BLOCK_SIZE);
}

if (i != 0){
    MPI_Wait(&request,
             MPI_STATUS_IGNORE);
}
```

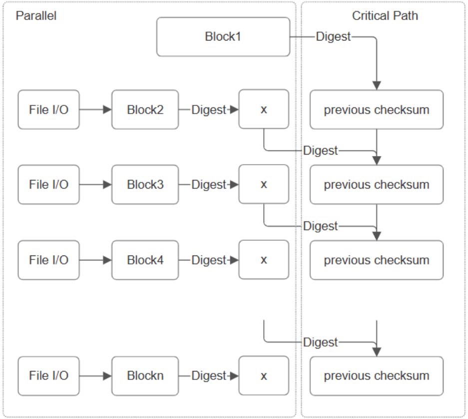
MPI Code(cont.)

Non-blocking send my checksum

```c
unsigned int len = 0;
for (int j=i; j<upper_bound; j++){
    EVP_DigestUpdate(ctx[j-i], prev_md,
            SHA512_DIGEST_LENGTH);
    EVP_DigestFinal_ex(ctx[j-i], prev_md, &len);
}
if (upper_bound != num_block) {
MPI_Isend(prev_md,
          SHA512_DIGEST_LENGTH,
          MPI_UINT8_T,
          recepient,
          0,
          MPI_COMM_WORLD,
          &request);
}
```

```
(hpc101) jrguo@M600:~/hpc101/sha512$ mpirun -n 1 baseline 2G1.bin baseline.out
"2G1.bin" size: 2147483648
checksum: 7224ec372ae2480f6609c35fe3ed6e1c7ea37f87a9d5e3e79b90038647eab9299db7d810a21acb7fb53884993869d5e160604e69ae5e459e5fbb3bf12c8
0b270
checksum time cost: 4230ms
total time cost: 22535ms
(hpc101) jrguo@M600:~/hpc101/sha512$ cp 2G.bin 2G2.bin
(hpc101) jrguo@M600:~/hpc101/sha512$ mpirun -n 8 mycode 2G2.bin mycode.out
"2G2.bin" size: 2147483648
checksum: 7224ec372ae2480f6609c35fe3ed6e1c7ea37f87a9d5e3e79b90038647eab9299db7d810a21acb7fb53884993869d5e160604e69ae5e459e5fbb3bf12c8
0b270
checksum time cost: 18401ms
total time cost: 18403ms
```

Wrap up



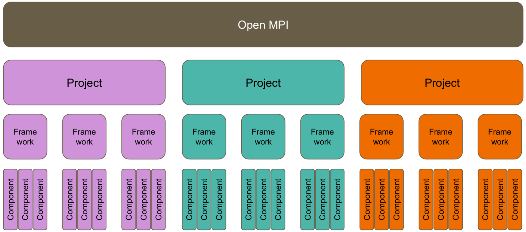
Parallel
Critical Path
Block1
Digest
File I/O
Block2
Digest
x
previous checksum
Digest
File I/O
Block3
Digest
x
previous checksum
Digest
File I/O
Block4
Digest
x
previous checksum
Digest
File I/O
Blockn
Digest
x
previous checksum

Miscellaneous

Open MPI

Modular Component Architecture(MCA)

- MCA framework
- MCA component
- MCA module



Open MPI Overall Architecture Terminology

Open MPI

3 Types of Open MPI Framwork

- In the MPI layer (OMPI)
- In the run-time layer (ORTE)
- In the operating system/platform layer (OPAL)

You might think of these frameworks as ways to group MCA parameters by function. (e.g. btl in OMPI)

```
~$ ompi_info --param btl all
      MCA btl: vader (MCA v2.1.0, API v3.1.0, Component v4.1.6)
      MCA btl: self (MCA v2.1.0, API v3.1.0, Component v4.1.6)
      MCA btl: tcp (MCA v2.1.0, API v3.1.0, Component v4.1.6)
  MCA btl tcp: ---------------------------------------------------
  MCA btl tcp: parameter "btl_tcp_if_include" (current value: "",
               data source: default, level: 1 user/basic, type:
               string)
               Comma-delimited list of devices and/or CIDR
               notation of networks to use for MPI communication
               (e.g., "eth0,192.168.0.0/16").  Mutually exclusive
               with btl_tcp_if_exclude.
  MCA btl tcp: parameter "btl_tcp_if_exclude" (current value:
               "127.0.0.1/8,sppp", data source: default, level: 1
               user/basic, type: string)
               Comma-delimited list of devices and/or CIDR
               notation of networks to NOT use for MPI
               communication -- all devices not matching these
               specifications will be used (e.g.,
               "eth0,192.168.0.0/16").  If set to a non-default
               value, it is mutually exclusive with
               btl_tcp_if_include.
  MCA btl tcp: parameter "btl_tcp_progress_thread" (current value:
               "0", data source: default, level: 1 user/basic,
               type: int)
```

ompi_info

Open MPI(Installation) cont.

Specify Compilers

./configure CC=/path/to/clang

CXX=/path/to/clang++ FC=/path/to/gfortran ...

Static or Shared ?

- –enable-static / –disable-static (default)
  libmpi.a
- –enable-shared / –disable-shared
  libmpi.so

Communication Library

UCX (Unified Communication X)

–with-ucx[=UCX_INSTALL_DIR]

With CUDA support

./configure –with-cuda[=/path/to/cuda]

Open MPI(mpirun)

- -x [env]

  Passes environment variables to remote nodes.
- –bind-to core
- -hostfile [hostfile]
- ...

Loading MPI On Our Cluster

- $ module load openmpi/5.0.3-pe46zvn
- $ module load intel-oneapi-mpi/2021.13.0-hpxfbao

Profiling and Tuning MPI Programs

Later lectures. (7.12)

Thank You

Questions?

References

https://rookiehpc.org/mpi/docs/index.html

https://docs.open-mpi.org/en/v5.0.x/index.html

https://www.intel.com/content/www/us/en/docs/mpi-library/developer-reference-linux/2021-13/overview.html