The Message Passing Interface (MPI): Parallelism on Distributed CPUs

http://mpi-forum.org
https://www.open-mpi.org/

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Why Two URLs?

http://mpi-forum.org
This is the definitive reference for the MPI standard. Go here if you want to read the official specification, which, BTW, continues to evolve.

https://www.open-mpi.org/
This consortium formed later. This is the open source version of MPI. If you want to start using MPI, I recommend you look here. This is the MPI that the COE systems use.

https://www.open-mpi.org/doc/v4.0/
This URL is also really good – it is a link to all of the MPI man pages.
MPI: The Basic Idea

Programs on different CPUs coordinate computations by passing messages between each other.

Note: Each CPU in the MPI "cluster" must be prepared ahead of time by having the MPI server code installed on it. Each MPI CPU must also have an integer ID assigned to it (called its rank).
This paradigm is how modern supercomputers work!

The Texas Advanced Computing Center’s Frontera supercomputer

How to SSH to the COE MPI Cluster

```
flip3 151% ssh submit-c@hpc.engr.oregonstate.edu

submit-c 142% module load slurm
submit-c 143% module load openmpi
```

Type these two lines right away to set your paths correctly

BTW, you can find out more about the COE cluster here:

https://it.engineering.oregonstate.edu/hpc

"The College of Engineering HPC cluster is a heterogeneous mix of 180 servers providing nearly 4000 CPU cores, over 140 GPUs, and over 36 TB total RAM. The systems are connected via gigabit ethernet, and most of the latest servers also utilize a Mellanox EDR InfiniBand network connection. The cluster also has access to 100TB global scratch from the College of Engineering's Dell/EMC Isilon enterprise storage. The CoE HPC Cluster is rated at over 900 peak TFLOPS (double-precision)."
### Compiling and Running

```
mpicc -o program program.c

or

mpic++ -o program program.cpp
```

```
mpiexec -mca btl self,tcp -np 4 program
```

All distributed processors execute the same program at the same time.

**# of processors to use**

**Warning – use mpic++ and mpiexec!**

Don’t use g++ and don’t run by just typing the name of the executable!

### Running with a bash Batch Script

**submit.bash:**

```bash
#!/bin/bash
#SBATCH -J AutoCorr
#SBATCH -A cs475-575
#SBATCH -p classmpitest
#SBATCH -N 4      # number of nodes
#SBATCH -n 4      # number of tasks
#SBATCH --constraint=ib
#SBATCH -o autocorr.out
#SBATCH -e autocorr.err
#SBATCH --mail-type=END,FAIL
#SBATCH --mail-user=joeparallel@cs.oregonstate.edu
module load openmpi
mpic++ autocorr.cpp -o autocorr -lm
mpiexec -mca btl self,tcp -np 4 ./autocorr
```

This is the partition name that we use for our class when testing your program. Use classmpifinal for taking your final performance numbers.

**submit-c 143% sbatch submit.bash**

Submitted batch job 258759
What is the Difference Between the Partitions `classmpitest` and `classmpifinal`?

`classmpitest` lets your program get into the system sooner, but it might be running alongside other jobs, so its performance might suffer. But, you don’t care because you are just compiling and debugging, not taking performance numbers for your report.

`classmpifinal` makes your program wait in line until it can get dedicated resources so that you get performance results that are much more representative of what the machines can do, and thus are worthy to be listed in your report.

Auto-Notifications via Email

`#SBATCH --mail-user=joeparallel@oregonstate.edu`

You don’t have to ask for email notification, but if you do, please, please, please be sure you get your email address right!

The IT people are getting real tired of fielding the bounced emails when people spell their own email address wrong.
Use slurm's `scancel` if your Job Needs to Be Killed

submit-c 143% `sbatch submit.bash`
Submitted batch job 258759

submit-c 144% `scancel 258759`

---

Setting Up and Finishing MPI

```c
#include <mpi.h>

int main(int argc, char *argv[])
{
  ...
  MPI_Init(&argc, &argv);
  ...
  MPI_Finalize();
  return 0;
}
```

You don’t need to process command line arguments if you don’t need to. You can also call it as:

```c
MPI_Init(NULL, NULL);
```
A **communicator** is a collection of CPUs that are capable of sending messages to each other. Getting information about our place in the communicator:

```c
int numCPUs;  // total # of cpus involved
int me;       // which one I am

MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_rank( MPI_COMM_WORLD, &me );
```

It is then each CPU's job to figure out what piece of the overall problem it is responsible for and then go do it.

MPI Follows a Single-Program-Multiple-Data (SPMD) Model

This requires MPI server code getting installed on all those CPUs. Only an administrator can do this.

A First Test of MPI

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

#define BOSS 0

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    int numCPUs;  // total # of cpus involved
    int me;       // which one I am

    MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );
    MPI_Comm_rank( MPI_COMM_WORLD, &me );

    if( me == BOSS )
        fprintf( stderr, "Rank %d says that we have a Communicator of size %d
        else
            fprintf( stderr, "Welcome from Rank %d

    MPI_Finalize( );
    return 0;
}
```
So, we have a group (a “communicator”) of distributed processors. How do they communicate about what work they are supposed to do?

Who am I? Where am I? What am I supposed to be doing? Hello? Is anyone listening?

Example: You could coordinate the units of our DGX system using MPI
A Good Place to Start:
MPI Broadcasting

MPI_Bcast( array, count, type, src, MPI_COMM_WORLD );

Both the sender and receivers need to execute MPI_Bcast – there is no separate receive function

MPI Broadcast Example

This is our heat transfer equation from before. Clearly, every CPU will need to know this value.

\[ \Delta T_i = \left( \frac{k}{\rho C} \right) \left( \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2} \right) \Delta t \]

int numCPUs;  
int me;  
float k_over_rho_c;  // the BOSS node will know this value, the others won’t (yet)

#define BOSS 0  
MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );  // how many are in this communicator  
MPI_Comm_rank( MPI_COMM_WORLD, &me );  // which one am I?

if( me == BOSS )  
{  
  <<< read k_over_rho_c from the data file >>>
}

MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD );  // send if BOSS, and receive if not
Confused? Look at this Diagram

Both the sender and receivers need to execute `MPI_Bcast` – there is no separate receive function.

Node #BOSS:

```c
MPI_Bcast(&k_over_rho_c, MPI_FLOAT, BOSS, MPI_COMM_WORLD); // send if BOSS, and receive if not
```

All Nodes that are not #BOSS:

- Executable code: `k_over_rho_c (being set)`
- Executable code: `k_over_rho_c (being set)`
- Executable code: `k_over_rho_c (being set)`
- Executable code: `k_over_rho_c (being set)`

How Does this Work?
Think Star Trek Wormholes!

![Wormhole Image](image-url)
### Sending Data from One Source CPU to Just One Destination CPU

```c
MPI_Send( array, numToSend, type, dst, tag, MPI_COMM_WORLD );
```

- **address of data to send from**
- **# elements**
  - (note: this is the number of elements, not the number of bytes!)
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - • • •
- **rank of the CPU to send to**
- **An integer or character to differentiate this transmission from any other transmission. I like to use chars.**

**Rules:**
- One message from a specific `src` to a specific `dst` cannot overtake a previous message from the same `src` to the same `dst`.
- `MPI_Send( )` blocks until the transfer is far enough along that `array` can be destroyed or re-used.
- There are no guarantees on order from different `src`'s.

---

### Receiving Data in a Destination CPU from a Source CPU

```c
MPI_Recv( array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status );
```

- **address of data to receive into**
- **# elements we can receive, at most**
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - • • •
- **Rank of the CPU we are expecting to get a transmission from**
- **Type = MPI_Status**
- **An integer or character to differentiate what transmission we are looking for with this call (be sure this matches what the sender is sending!). I like to use chars.**

**Rules:**
- The receiver blocks waiting for data that matches what it declares to be looking for
- One message from a specific `src` to a specific `dst` cannot overtake a previous message from the same `src` to the same `dst`
- There are no guarantees on the order from different `src`'s
- The order from different `src`'s could be implied in the `tag`
- `status` is type MPI_Status – the "&status" can be replaced with MPI_STATUS_IGNORE
Example

Remember, this identical code runs on all CPUs:

```c
int numCPUs;
int me;
#define MYDATA_SIZE 128
char mydata[MYDATA_SIZE];
#define BOSS 0

MPI_Comm_size(MPI_COMM_WORLD, &numCPUs);
MPI_Comm_rank(MPI_COMM_WORLD, &me);

if( me == BOSS ) // the primary
{
  for( int dst = 0; dst < numCPUs; dst++ )
  {
    if( dst != BOSS )
    {
      char *InputData = "Hello, Beavers!";
      MPI_Send( InputData, strlen(InputData) + 1, MPI_CHAR, dst, 'B', MPI_COMM_WORLD );
    }
  }
}
else // a secondary
{
  MPI_Recv( myData, MYDATA_SIZE, MPI_CHAR, BOSS, 'B', MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  printf( " '%s' from rank # %d
", in, me );
}
```

Example

Look at this Diagram

**Source**

Executable code

MyData

Destinations

Executable code

Input Data

Executable code

MyData

Executable code

MyData

Executable code

MyData

You are highly discouraged from sending to yourself. Because both the send and receive are capable of blocking, the result could be deadlock.
How does MPI let the Sender perform an MPI_Send() even if the Receivers are not ready to MPI_Recv()?

**MPI_Send()** blocks until the transfer is far enough along that the array can be destroyed or re-used.

```
#define NUMELEMENTS ???
int numCPUs;
int me;
#define BOSS 0
MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_rank( MPI_COMM_WORLD, &me );
int PPSize = NUMELEMENTS / numCPUs; // per-processor data size -- assuming it comes out evenly
float *myData = new float [ PPSize ];

if( me == BOSS ) // the sender
{
    float *InputData = new float [ NUMELEMENTS ];
    << read the full input data into InputData from disk >>
    for( int dst = 0; dst < numCPUs; dst++ )
    {
        if( dst != BOSS )
        {
            MPI_Send( &InputData[dst*PPSize], PPSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD );
        }
    }
}
else // a receiver
{
    MPI_Recv( myData, PPSize, MPI_FLOAT, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    // do something with this subset of the data
}
```

Another Example

You typically don’t send the entire workload to each dst – you just send part of it, like this:

```cpp
#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numCPUs);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    int PPSize = NUMELEMENTS / numCPUs; // per-processor data size -- assuming it comes out evenly
    float *myData = new float [ PPSize ];
    if( me == BOSS ) // the sender
    {
        float *InputData = new float [ NUMELEMENTS ];
        << read the full input data into InputData from disk >>
        for( int dst = 0; dst < numCPUs; dst++ )
        {
            if( dst != BOSS )
            {
                MPI_Send( &InputData[dst*PPSize], PPSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD );
            }
        }
    }
    else // a receiver
    {
        MPI_Recv( myData, PPSize, MPI_FLOAT, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
        // do something with this subset of the data
    }
    MPI_Finalize();
    return 0;
}
```
You typically don’t send the entire workload to each dst – you just send part of it, like this:

```c
if ( dst != BOSS ) {
    MPI_Send( &inputData[dst*PPSize], PPSize, MPI_FLOAT, dst, MPI_COMM_WORLD );
}

MPI_Recv( myData, PPSize, MPI_FLOAT, BOSS, MPI_COMM_WORLD, NULL, MPI_STATUSES_IGNORE );
```

In Distributed Computing, You Often Hear About These Design Patterns

- **Broadcast**
- **Scatter**
- **Gather**
Scatter and Gather Usually Go Together

Note surprisingly, this is referred to as Scatter/Gather

MPI Scatter

Take a data array, break it into ~equal portions, and send it to each CPU

Both the sender and receivers need to execute MPI_Scatter. There is no separate receive function
Both the sender and receivers need to execute **MPI_Gather**.
There is no separate receive function.

**Remember This? It’s Baaaaaack as a complete Scatter/Gather Example**

The **Compute : Communicate Ratio** still applies, except that it is even more important now because there is much more overhead in the Communicate portion.

This pattern of breaking a big problem up into pieces, sending them to different CPUs, computing on the pieces, and getting the results back is very common. That’s why MPI has its own scatter and gather functions.
#include <stdio.h>
#include <math.h>
#include <mpi.h>

const float RHO = 8050.;
const float C   = 0.466;
const float K   = 20.;
float k_over_rho_c =  K / (RHO*C); // units of m^2/sec

// K / (RHO*C) = 5.33x10^-6 m^2/sec

const float DX    = 1.0;
const float DT    = 1.0;

#define BOSS 0
#define NUMELEMENTS (8*1024*1024)
#define NUM_TIME_STEPS 4
#define DEBUG false

float * NextTemps; // per-processor array to hold computer next-values
int NumCpus; // total # of cpus involved
int PPSize; // per-processor local array size
float * PPTemps; // per-processor local array temperature data
float * TempData; // the overall NUMELEMENTS-big temperature data

void DoOneTimeStep( int );

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    int me; // which one I am

    MPI_Comm_size( MPI_COMM_WORLD, &NumCpus );
    MPI_Comm_rank( MPI_COMM_WORLD, &me );

    // decide how much data to send to each processor:
    PPSize = NUMELEMENTS / NumCpus; // assuming it comes out evenly
    PPTemps = new float [PPSize]; // all processors now have this uninitialized Local array
    NextTemps = new float [PPSize]; // all processors now have this uninitialized local array too

    // broadcast the constant:
    MPI_Bcast( (void *)&k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD );

    DoOneTimeStep( me );

    return 0;
}
if ( me == BOSS )  // this is the data-creator
{
    TempData = new float [NUMELEMENTS];
    for ( int i = 0; i < NUMELEMENTS; i++ )
        TempData[ i ] = 0.;
    TempData[NUMELEMENTS/2] = 100.;
}
MPI_Scatter( TempData, PPSize, MPI_FLOAT, PPTemps, PPSize, MPI_FLOAT,
            BOSS, MPI_COMM_WORLD );

// all the PPTemps arrays have now been filled
// do the time steps:
    double time0 = MPI_Wtime();
    for( int steps = 0; steps < NUM_TIME_STEPS; steps++ )
    {
        // do the computation for one time step:
        DoOneTimeStep( me );
    }
// ask for all the data:
#ifdef WANT_EACH_TIME_STEPS_DATA
    MPI_Gather( PPTemps, PPSize, MPI_FLOAT, TempData, PPSize, MPI_FLOAT,
                BOSS, MPI_COMM_WORLD );
#endif
#endif
#ifdef WANT_EACH_TIME_STEPS_DATA
    MPI_Gather( PPTemps, PPSize, MPI_FLOAT, TempData, PPSize, MPI_FLOAT,
                BOSS, MPI_COMM_WORLD );
#endif
    double time1 = MPI_Wtime();
if( me == BOSS )
{
    double seconds = time1 - time0;
    double performance =
        (double)NUM_TIME_STEPS * (double)NUMELEMENTS / seconds / 1000000.;
    // mega-elements computed per second
    fprintf( stderr, "%3d, %10d, %8.2lf\n", NumCpus, NUMELEMENTS, performance );
}

MPI_Finalize( );
return 0;

// read from PerProcessorData[ ], write into NextTemps[ ]
void
DoOneTimeStep( int me )
{

    MPI_Status status;
    // send out the left and right end values:
    // (the tag is from the point of view of the sender)
    if( me != 0 ) // i.e., if i'm not the first group on the left
    {
        // send my PPTemps[0] to me-1 using tag 'L'
        MPI_Send( &PPTemps[0], 1, MPI_FLOAT, me-1, 'L', MPI_COMM_WORLD );
        if( DEBUG ) fprintf( stderr, "%3d sent 'L' to %3d\n", me, me-1 );
    }
    
    if( me != NumCpus-1 ) // i.e., not the last group on the right
    {
        // send my PPTemps[PPSize-1] to me+1 using tag 'R'
        MPI_Send( &PPTemps[PPSize-1], 1, MPI_FLOAT, me+1, 'R', MPI_COMM_WORLD );
        if( DEBUG ) fprintf( stderr, "%3d sent 'R' to %3d\n", me, me+1 );
    }
DoOneTimeStep, II

float left = 0.;
float right = 0.;

if( me != 0 ) // i.e., if i'm not the first group on the left
{
    // receive my "left" from me-1 using tag 'R'
    MPI_Recv( &left, 1, MPI_FLOAT, me-1, 'R', MPI_COMM_WORLD, &status );
    if( DEBUG )    fprintf( stderr, "%3d received 'R' from %3d
", me, me-1 );
}

if( me != NumCpus-1 ) // i.e., not the last group on the right
{
    // receive my "right" from me+1 using tag 'L'
    MPI_Recv( &right, 1, MPI_FLOAT, me+1, 'L', MPI_COMM_WORLD, &status );
    if( DEBUG )    fprintf( stderr, "%3d received 'L' from %3d
", me, me+1 );
}

Sharing Values Across the Boundaries

1 sent 'L' to   0
1 sent 'R' to   2
2 sent 'L' to   1
2 sent 'R' to   3
2 received 'R' from   1
0 sent 'R' to   1
0 received 'L' from   1
1 received 'R' from   0
1 received 'L' from   2
3 sent 'L' to   2
3 received 'R' from   2
2 received 'L' from   3
1D Compute-to-Communicate Ratio

Intraprocessor computing

Interprocessor communication

Compute : Communicate ratio = N : 2

where N is the number of compute cells per processor

In the above drawing, Compute : Communicate is 4 : 2

DoOneTimeStep, III

// first element on the left (0):
{
    float dtemp = (k_over_rho_c * 
         ( left - 2.*PPTemps[0] + PPTemps[1] ) / ( DX*DX ) ) * DT;
    NextTemps[0] = PPTemps[0] + dtemp;
}

// all the nodes in the middle:
for( int i = 1; i < PPSize-1; i++ )
{
    float dtemp = (k_over_rho_c * 
         ( PPTemps[i-1] - 2.*PPTemps[i] + PPTemps[i+1] ) / ( DX*DX ) ) * DT;
    NextTemps[i] = PPTemps[i] + dtemp;
}

// last element on the right (PPSize-1):
{
    float dtemp = (k_over_rho_c * 
         ( PPTemps[PPSize-2] - 2.*PPTemps[PPSize-1] + right ) / ( DX*DX ) ) * DT;
    NextTemps[PPSize-1] = PPTemps[PPSize-1] + dtemp;
}
// update the local dataset:
for( int i = 0; i < PPSize; i++ )
{
    PPTemps[ i ] = NextTemps[ i ];
}

### MPI Performance

**Mega-Elements Computed Per Second vs. Number of Elements**

- **Number of Elements**
- **Mega-Elements Computed Per Second**
- **Number of Processors**
From: Peter Johnsen, Mark Straka, Melvyn Shapiro, Alan Norton, Thomas Galarneau, Petascale WRF Simulation of Hurricane Sandy.

Using MPI and OpenMP on 13,680 nodes (437,760 cores) of the Cray XE6 at NCSA at the University of Illinois

**MPI Reduction**

\[ \text{MPI\_Reduce}( \text{partialResult}, \text{globalResult}, \text{count}, \text{type}, \text{operator}, \text{dst}, \text{MPI\_COMM\_WORLD}); \]

- Where the partial result is stored on each CPU
- Place to store the full result on the dst CPU
- Number of elements in the partial result
- MPI\_CHAR
- MPI\_INT
- MPI\_LONG
- MPI\_FLOAT
- MPI\_DOUBLE
- \*\*\*
- Who is given the final answer
- MPI\_MIN
- MPI\_MAX
- MPI\_SUM
- MPI\_PROD
- MPI\_MINLOC
- MPI\_MAXLOC
- MPI\_LAND
- MPI\_BAND
- MPI\_LOR
- MPI\_BOR
- MPI\_LXOR
- MPI\_BXOR

This really should be called **Scatter/Gather/Reduction**

Both the sender and receivers need to execute \text{MPI\_Reduce}. There is no separate receive function.
MPI Reduction Example

// gratuitous use of a reduce -- average all the temperatures:

float partialSum = 0.;
for( int i = 0; i < PPSize; i++ )
    partialSum += PPTemps[ i ];

float globalSum = 0.;
MPI_Reduce( &partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, BOSS, MPI_COMM_WORLD );

if( me == BOSS )
    fprintf( stderr, "Average temperature = %f\n", globalSum/(float)NUMELEMENTS );

MPI Barriers

MPI_Barrier( MPI_COMM_WORLD );

All CPUs must execute the call to MPI_Barrier() before any of the CPUs can move past it. That is, each CPU’s MPI_Barrier() blocks until all CPUs execute a call to MPI_Barrier().
MPI Derived Types

Idea: In addition to types MPI_INT, MPI_FLOAT, etc., allow the creation of new MPI types so that you can transmit an “array of structures”.

Reason: There is significant overhead with each transmission. Better to send one entire array of structures instead of sending several arrays separately.

```c
MPI_Type_create_struct( count, blocklengths, displacements, types, datatype );
```

```c
struct point
{
  int pointSize;
  float x, y, z;
};
```

```c
MPI_Datatype MPI_POINT;
int blocklengths[] = { 1, 1, 1, 1 ];
int displacements[] = { 0, 4, 8, 12 };;
MPI_type types[] = { MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT };
MPI_Type_create_struct( 4, blocklengths, displacements, types, &MPI_POINT );
```

You can now use MPI_POINT everywhere you could have used MPI_INT, MPI_FLOAT, etc.

---

MPI Timing

```c
double MPI_Wtick();
```

Returns the resolution of the clock, in seconds.

```c
double MPI_Wtime();
```

Returns the time, in seconds, since “some time in the past”.

**Warning:** the clocks on the different CPUs are not guaranteed to be synchronized!
MPI Status-Checking

Some MPI calls have a `&status` in their argument list.

The `status` argument is declared to be of type `MPI_Status`, which is defined as this struct:

```c
typedef struct  _MPI_Status
{
    int  MPI_SOURCE;
    int  MPI_TAG;
    int  MPI_ERROR;
} MPI_Status;
```

- `MPI_SOURCE` is the rank of the node who sent this
- `MPI_TAG` is the tag used during the send
- `MPI_ERROR` is the error number that occurred

Example:

```c
MPI_Status status;
MPI_Recv( myData, MYDATA_SIZE, MPI_CHAR, BOSS, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
```

fprintf( stderr, "Rank = %d, Tag = %d, Error Code = %d\n",
    status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR );

MPI Error Codes

- No error
- Invalid buffer pointer
- Invalid count argument
- Invalid datatype argument
- Invalid tag argument
- Invalid communicator
- Invalid rank
- Invalid request (handle)
- Invalid next
- Invalid group
- Invalid operation
- Invalid topology
- Invalid dimension argument
- Invalid argument of some other kind
- Unknown error Message transmitted on receive
- Known error not in this list
- Internal MPI (implementation) error
- Error code in status
- Pending request
- Invalid keyval has been passed
- Invalid base passed to MPI_FREE_MEM
- Key larger than MPI_MAX_INFO_KEY
- Value larger than MPI_MAX_INFO_VAL
- Invalid key passed to MPI_INFO_DELETE
- Error in opening process
- Invalid port name passed to MPI_COMM_CONNECT
- Invalid service name passed to MPI_UNREGISTER_NAME
- Invalid service name passed to MPI_LOOKUP_NAME
- Invalid win argument
- Invalid state argument
- Invalid disp argument
- Invalid info argument
- Invalid locktype argument
- Invalid assert argument
- Conflicting access to window
- Wrong synchronization of RMA calls

Collective argument not identical on all processes, or collective routines called in a different order by different processes.

Error related to file opened to `MPI_FILE_OPEN`.

Unsupported operation passed to `MPI_FILE_SET_VIEW`.

Unsupported operation, such as seeking on a file which supports sequential access only

File does not exist

File exists

Invalid file name (e.g., path name too long)

Permission denied

Not enough space

Quota exceeded

Read only file or file system

Conversion functions could not be registered because a data representation identifier that was already defined was passed to `MPI_REGISTER_DATAREP`.

An error occurred to a user supplied data conversion function.

Other I/O error

Last error code