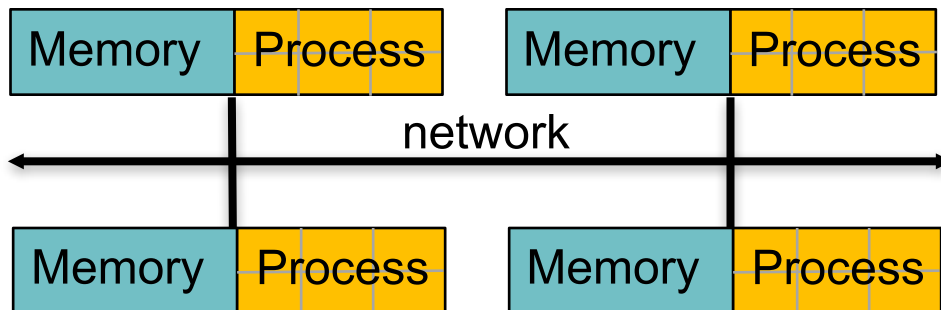
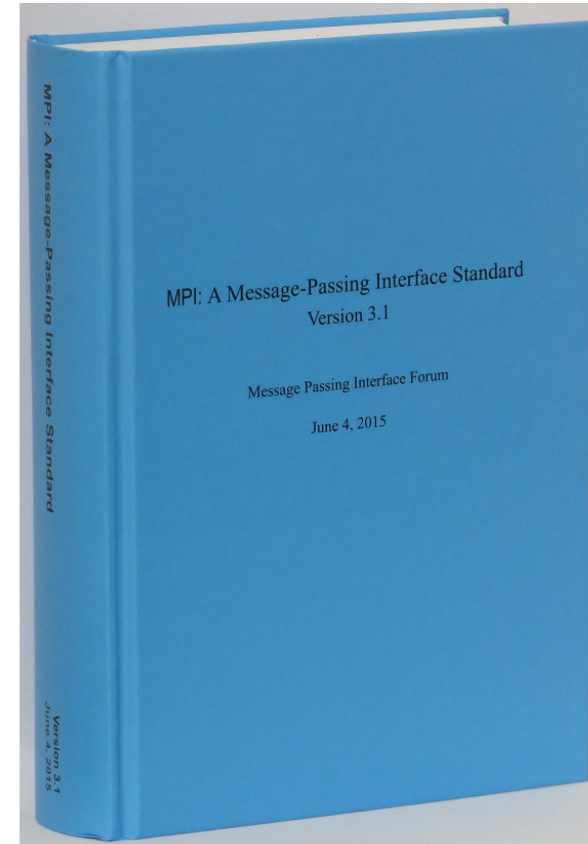
Message Passing Interface

George Bosilca
bosilca@icl.utk.edu

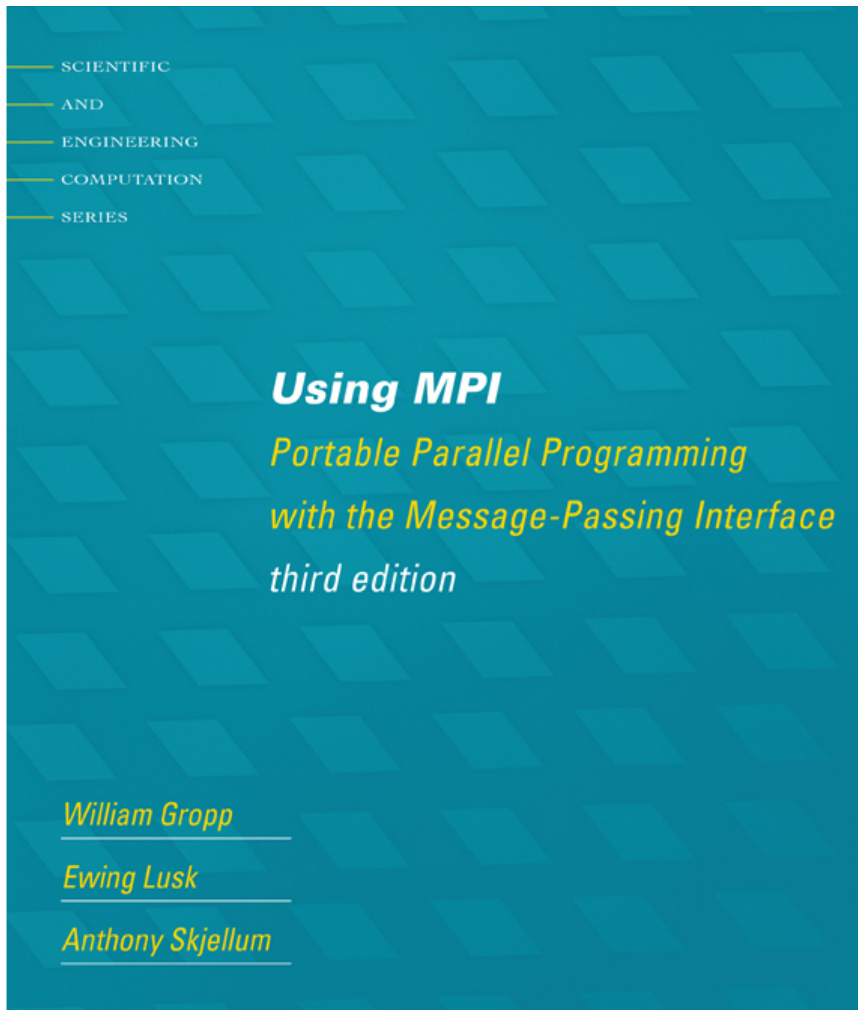
Message Passing Interface Standard

- http://www.mpi-forum.org
- Current version: 3.1
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

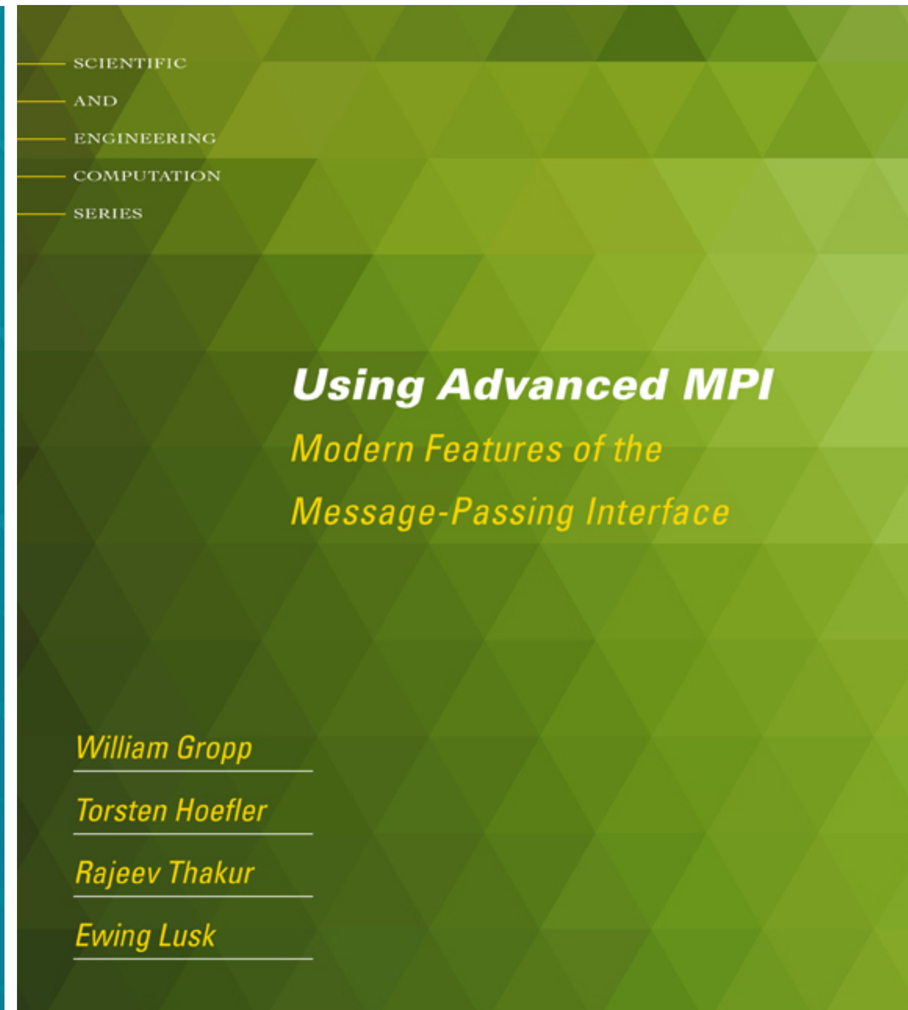




For more info: Books on MPI



Basic MPI



Advanced MPI, including MPI-3

Even more info

- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/
- Implementations: Open MPI, MPICH, MVAPICH, Intel MPI, Microsoft MPI, and others

Why MPI is important ?

- Standardization - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries
- Portability - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance
- Functionality – Rich set of features
- Availability - A variety of implementations are available, both vendor and public domain

Before you start

- Compiling and linking MPI application is as simple as calling mpicc, mpicxx and mpifort
  - Your MPI library must be configured with options similar to your applications (-i8)
- Executing MPI application is slightly more complicated
  - mpiexec –np 8 my_awesome_app
  - For more advanced usages carefully read the ”mpiexec –help” and/or the online resources

MPI Initialization/Finalization

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

Setting up your world

- MPI can be used between MPI_Init and MPI_Finalize
  - Predefined communication contexts exists after MPI_Init
    - MPI_COMM_WORLD: all processes created by the same mpiexec command
    - MPI_COMM_SELF: you are the only participant
  - These predefined objects lose their validity after MPI_Finalize (as everything else related to MPI)

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize( void)
int MPI_Initialized(int* flag)
int MPI_Finalized(int* flag)
```

Threading support

- Request a certain level of thread support
  - There is a certain cost involved

MPI_THREAD_SINGLE	Only the thread executes in the context of the process (the process should not be multi-threaded)
MPI_THREAD_FUNNELED	Only the thread that called MPI_Init_thread can make MPI calls (the process can be multi-threaded)
MPI_THREAD_SERIALIZED	Only one thread at the time will make MPI calls (the process can be multi-threaded)
MPI_THREAD_MULTIPLE	All threads are allowed to make MPI calls simultaneously (the process can be multi-threaded)

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int* provided)
```

Hello World !

```
int main(int argc, char** argv) {
    int world_size, world_rank, name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hello world from %s, rank %d/%d\n",
            processor_name, world_rank, world_size);

    MPI_Finalize();
}
```

MPI Point-to-point communications

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

Send & Receive

- Explicit communications (FIFO per peer per communicator)
- Move data from one process to another (possibly local) process
  - The data is described by a data-type, a count and a memory location
  - The destination process by a rank in a communicator
  - The matching is tag based

```
int MPI_Send( void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status* status)
```

Blocking Communications

- The process is blocked in the MPI function until:
  - For receives the remote data has been safely copied into the receive buffer
  - For sends the send buffer can be safely modified by the user without impacting the message transfer

```
int MPI_Send( void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status* status)
```

Communication modes

- a send in **Standard** mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
  - successful completion of the send operation may depend on the occurrence of a matching receive
- **Buffered** mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted.
  - its completion does not depend on the occurrence of a matching receive
- send that uses the **Synchronous** mode can be started whether or not a matching receive was posted. It will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message
  - Its completion does not only indicates that the send buffer can be reused, but it also indicates that the receiver started executing the matching receive

Communication modes

- send that uses the **Ready** communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined.
  - completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused

	Buffered	Synchronous	Ready
Send	MPI_Bsend	MPI_Ssend	MPI_Rsend

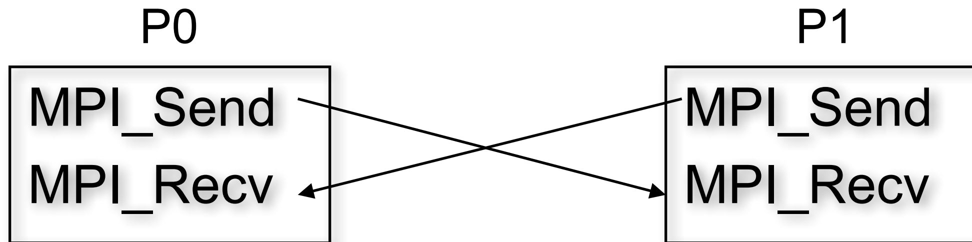
Semantics of Point-to-Point Communication

- Order: Messages are non-overtaking
- Progress: No progression guarantees except when in MPI calls
- Fairness: no guarantee of fairness. However, usually a best effort approach implemented in the MPI libraries.
- Resource limitations: Best effort

Quality implementation: a particular implementation of he standard, exhibiting a set of desired properties.

Don’t do !

```
MPI_Send( buf, count, datatype, peer, tag, comm)
MPI_Recv( buf, count, datatype, peer, tag, comm, &status)
```



Non-Blocking Communications

- The process returns from the call as soon as possible, before any data transfer has been initiated.
- All flavors of communication modes supported.
- Subsequent MPI call required to check the completion status.

```
int MPI_Isend( void* buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm,
               MPI_Request *request )
int MPI_Irecv( void* buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request )
```

Communication Completion

- Single completion
  - completion of a send operation indicates that the sender is now free to update the locations in the send buffer
  - completion of a receive operation indicates that the receive buffer contains the received message

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
int MPI_Test( MPI_Request *request,
                    int *flag, MPI_Status *status)
```

Communication Completion

- Multiple Completions (ANY)
  - A call to MPI_WAITANY or MPI_TESTANY can be used to wait for the completion of one out of several operations.

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )
int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```

Communication Completion

- Multiple Completions (SOME)
  - A call to MPI_WAITSOME or MPI_TESTSOME can be used to wait for the completion of at least one out of several operations.

```
int MPI_Waitsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
int MPI_Testsome( int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses )
```

Communication Completion

- Multiple Completions (ALL)
  - A call to MPI_WAITALL or MPI_TESTALL can be used to wait for the completion of all operations.

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )
int MPI_Testall( int count, MPI_Request *array_of_requests,
                 int *flag, MPI_Status *array_of_statuses )
```

Persistent Communications

- A communication with the same argument list repeatedly executed within the inner loop of a parallel computation
  - Allow MPI implementations to optimize the data transfers
- All communication modes (buffered, synchronous and ready) can be applied

```
int MPI_[B,S, R,]Send_init( void* buf, int count, MPI_Datatype datatype,
                            int dest, int tag, MPI_Comm comm,
                            MPI_Request *request )
int MPI_Recv_init( void* buf, int count, MPI_Datatype datatype,
                            int source, int tag, MPI_Comm comm,
                            MPI_Request *request )
int MPI_Start( MPI_Request *request )
int MPI_Startall( int count, MPI_Request *array_of_requests )
```