Introduction to the MPI programming model

Janko Strassburg

PATC Parallel Programming Workshop October 2013
Message Passing Interface – MPI
- De facto standard
- Although not an official standard (IEEE, …)

Maintained by the MPI forum
- Writes and ratifies the standard
- Consisting of academic, research and industry representatives
What is message passing?

Data transfer plus synchronization

- Requires cooperation of sender and receiver
- Cooperation not always apparent in code
Purpose

- Divide problems in parallel computing
- Use multiple processors, hundreds or thousands at a time
- Spread job across multiple nodes
  - Computations too big for one server
  - Memory requirements
  - Splitting the problem, divide and conquer approach
MPI Use

Abstracts view of the network
- Shared memory / Sockets
- Ethernet / Infiniband
- High speed communication network / High throughput data network
- ...

Application communicates using simple commands
- MPI_SEND / MPI_RECV
- Implementation handles connections automatically
Layered system
Abstracting hardware from the programmer
Example: 4 Nodes

- User application
- MPI API
- Operating System

Network
Example: 2 Nodes
Example: 1 Node

All processes running on the same machine
Parallel Programming

Possibilities:

- Dedicated parallel programming languages
- Extensions of existing sequential programming languages
- Usage of existing sequential programming languages + libraries with external functions for message passing
Usage of Existing Sequential Programming Languages

Approach:

– Use existing FORTRAN/C code
– Function calls to message passing library

Explicit parallelism:
User defines

– which processes to execute,
– when and how to exchange messages, and
– which data to exchange within messages.
**MPI Intention**

- Specification of a standard library for programming message passing systems

- Interface: practical, portable, efficient, and flexible

  - Easy to use

- For vendors, programmers, and users
MPI Goals

- Design of a standardized API
- Possibilities for efficient communication (Hardware-Specialities, ...)
- Implementations for heterogeneous environments
- Definition of an interface in a traditional way (comparable to other systems)
- Availability of extensions for increased flexibility
Collaboration of 40 Organisations world-wide:

- IBM T.J. Watson Research Center
- Intels NX/2
- Express
- nCUBE's VERTEX
- p4 - Portable Programs for Parallel Processors
- PARMACS
- Zipcode
- PVM
- Chameleon
- PICL
- ...

MPI Forum
Available Implementations

- **Open MPI:**
  - Combined effort from FT-MPI, LA-MPI, LAM/MPI, PACX-MPI
  - De facto standard; used on many TOP500 systems

- **MPICH**
- **CHIMP**
- **LAM**
- **FT-MPI**

- **Vendor specific implementations:**
  - Bull, Fujitsu, Cray, IBM, SGI, DEC, Parsytec, HP, ...
MPI Programming Model

**Parallelization:**
- Explicit parallel language constructs (for communication)
- Library with communication functions

**Programming Model:**
- SPMD (single program multiple data)
- All processes load the same source code
- Distinction through process number
**MPI Program**

**2 Parts:**
- User code
- MPI Functionality (from MPI Library)
MPI Functionality

- Process Creation and Execution
- Queries for system environment
- Point-to-point communication (Send/Receive)
- Collective Operations (Broadcast, ...)
- Process groups
- Communication context
- Process topologies
- Profiling Interface
Characteristics:

- For *Parallelism*, computation must be partitioned into multiple processes (or tasks)

- Processes are assigned to processors $\Rightarrow$ **mapping**
  - 1 Process $=$ 1 Processor
  - n Processes $=$ 1 Processor

- **Multitasking** on one processor:
  - Disadvantage: Longer execution time due to time-sharing
  - Advantage: Overlapping of communication latency
Granularity

The size of a process defines its granularity

- Coarse Granularity
  - each process contains many sequential execution blocks

- Fine Granularity
  - each process contains only few (sometimes one) instructions
Granularity

Granularity = 
Size of computational blocks between communication and synchronization operations

The higher the granularity, the
– smaller the costs for process creation
– smaller the number of possible processes and the achievable parallelism
Parallelization

- Data Partitioning:
  \[\text{SPMD} = \text{Single Program Multiple Data}\]

- Task Partitioning:
  \[\text{MPMD} = \text{Multiple Program Multiple Data}\]

Types of Process-Creation:
- Static
- Dynamic
Data Partitioning (SPMD)

**Implementation:**
1. Source code

```c
void main()
{
    int i, j;
    char a;
    for (i = 0; ...
```

**Execution:**
- n Executables

Process 1

```c
void main()
{
    int i, j;
    char a;
    for (i = 0; ...
```

Processor 1

Process 2

```c
void main()
{
    int i, j;
    char a;
    for (i = 0; ...
```

Processor 2

Process 3

```c
void main()
{
    int i, j;
    char a;
    for (i = 0; ...
```

Processor 3
Task-Partitioning (MPMD)

Implementation: m Source codes

Execution: n Executables

```
void main()
{
    int i,j;
    char a;
    for(i=0; ...

Processor 1
```

```
void main()
{
    int k;
    char b;
    while(b= ...

Processor 2
```

```
void main()
{
    int k;
    char b;
    while(b= ...

Processor 3
```

void main()
{
    int i,j;
    char a;
    for(i=0; ...

Processor 1
```

void main()
{
    int k;
    char b;
    while(b= ...

Processor 2
```

void main()
{
    int k;
    char b;
    while(b= ...

Processor 3
```

MPI - Message Passing Interface
Comparison: SPMD/MPMD

**SPMD:**

- One source code for all processes
- Distinction in the source code through control statements
  ```c
  if (pid() == MASTER) { ... }
  else { ... }
  ```
- Widely used
Comparison: SPMD/MPMD

MPMD:

- One source for each process
- Higher flexibility and modularity
- Administration of source codes difficult
- Additional effort during process creation
- Dynamic process creation possible
Process Creation

Static:

- All processes are defined before execution
- System starts a fixed number of processes
- Each process receives same copy of the code
**Dynamic:**

- Processes can creation/execute/terminate other processes during execution
- Number of processes changes during execution
- Special constructs or functions are needed

**Advantage**
- higher flexibility than SPMD

**Disadvantage**
- process creation expensive $\Rightarrow$ overhead
Process Creation/Execution

Commands:

- Creation and execution of processes is not part of the standard, but instead depends on the chosen implementation:
  
  **Compile:** `mpicc -o <exec> <file>.c`
  
  **Execute:** `mpirun -np <proc> <exec>`

- Process Creation: only static (before MPI-2)
- SPMD programming model
**Basic-Code-Fragment**

### Initialization and Exit:

1. `#include <mpi.h>`
2. `...`
3. `int main(int argc, char *argv[])`
4. `{`
5. `MPI_Init(&argc, &argv);`
6. `...`
7. `MPI_Finalize();`
8. `}`
Structure of MPI Functions

General:

1. result = MPI_Xxx(...);

Example:

1. result = MPI_Init(&argc, &argv);
2. if(result!=MPI_SUCCESS) {
3. fprintf(stderr,"Problem");
4. fflush(stderr);
5. MPI_Abort(MPI_COMM_WORLD,result);
6. }
Query Functions

Identification:

- **Who am I?**
  - Which process number has the current process?
    - `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)`

- **Who else is there?**
  - How many processes have been started?
    - `MPI_Comm_size(MPI_COMM_WORLD, &mysize)`

- Characteristics: 0 <= myrank < mysize
MPI & SPMD Restrictions

**Ideally:** Each process executes the same code.

**Usually:** One (or a few) processes execute slightly different codes.

**Preliminaries:**
Statements to distinguish processes and the subsequent code execution

**Example:** Master-slave program
⇒ complete code within one program/executable
Master-Slave Program

1. int main(int argc, char *argv[])
2. {
3.   MPI_Init(&argc, &argv);
4.   ...
5.   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6.   if(myrank == 0)
7.       master();
8.   else
9.       slave();
10.  ...
11.  MPI_Finalize();
12. }
Global Variables

Problem:
1. int main(int argc, char *argv[])
2. {
   float big_array[10000];

Solution:
1. int main(int argc, char *argv[])
2. {
3. if(myrank == 0) {
4.   float big_array[10000];

Allocate large arrays only on the ranks needed to save memory
Global Variables

Problem:
1. int main(int argc, char *argv[])
2. {
   float big_array[10000];
}

Solution:
1. int main(int argc, char *argv[])
2. {
3.   float *big_array;
4.   if(myrank == 0) {
5.     big_array = (float *)malloc(...)

If the other ranks need to know details about a variable, pointers can be created. The memory can be allocated dynamically within the correct rank.
Try to avoid communication as much as possible: more than a factor of 100/1000 between transporting a byte and doing a multiplication

– Often it is faster to replicate computation than to compute results on one process and communicate them to other processes.

Try to combine messages before sending.

– It is better to send one large message than several small ones.
Basic Functions:

- **send(parameter_list)**
- **recv(parameter_list)**

**Send Function:**
- In origin process
- Creates message

**Receive Function:**
- In destination process
- Receives transmitted message
Simple Functions

On the origin process:
\[
\text{send}(\&x, \text{destination} \_\text{id})
\]

On the destination process:
\[
\text{recv}(\&y, \text{source} \_\text{id})
\]

- Process 1:
  - Time
  - Message passing
  - Send(\&x, 2)

- Process 2:
  - Recv(\&y, 1)

MPI - Message Passing Interface
Standard Send

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

- **buf**: Address of message in memory
- **count**: Number of elements in message
- **datatype**: Data type of message
- **dest**: Destination process of message
- **tag**: Generic message tag
- **comm**: Communication handler

**MPI_Datatype**
- MPI_CHAR, MPI_INT, MPI_FLOAT, ...

**MPI_Comm**
- MPI_COMM_WORLD
Standard Receive

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

buf          Address of message in memory
count        Expected number of elements in message
datatype     Data type of message
source       Origin process of message
tag          Generic message tag
comm         Communication handler
status       Status-Information
Example: *Hello World*

```
Str = "Hello World"
Send(Str)
```

```
Recv(Str)
print(Str)
```
Example: *Hello World*

```c
1. if (myrank == 1) {
2.   char sendStr[] = "Hello World";
3.   MPI_Send(sendStr, strlen(sendStr)+1, MPI_CHAR,
4.       0, 3, MPI_COMM_WORLD);
5. }
6. else {
7.   char recvStr[20];
8.   MPI_Recv(recvStr, 20, MPI_CHAR, 1, 3,
9.       MPI_COMM_WORLD, &stat);
10.  printf("%s\n", recvStr);
11. }
```
Example: *Round Robin*

Process 0
- `MPI_Send(Str, ...)`
- `MPI_Recv(Str, ...)`
- `printf(Str)`

Process 1
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`

Process 2
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`

Process 3
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`
Standard Receive

Remark:
Maximum message length is fixed:

- If message is bigger ➔ overflow error
- If message is smaller ➔ unused memory

➔ Allocate sufficient space before calling MPI_Recv
Standard Receive

How many elements have been received?

```c
int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)
```

Status-Information:
1. `struct MPI_Status {`
2. `int MPI_SOURCE;`
3. `int MPI_TAG;`
4. `int MPI_ERROR;`
5. `int count;`
6. `...`
7. `};`

- Number of Elements
- in case of `MPI_ANY_SOURCE`
- in case of `MPI_ANY_TAG`
MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags
Communicators

Scope of processes

- Communicator group processes
- A group defines the set of processes, that can communicate with each other
- Used in point-to-point and collective communication
- After starting a program, its processes subscribe to the “Universe” —> MPI_COMM_WORLD
- Each program has its own “Universe”
Usage of Communicators

- Fence off communication environment
- Example: Communication in library
  *What happens, if a program uses a parallel library that uses MPI itself?*
- 2 Kinds of communicators:
  - Intra-communicator: inside a group
  - Inter-communicator: between groups
- Processes in each group are always numbered 0 to $m-1$ for $m$ processes in a group
MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags
Structure of Messages

**Standard data types:**
- Integer, Float, Character, Byte, ...
- (Continuous) arrays

**Complex data types:**
- Messages including different data: counter + elements
- Non-continuous data types: sparse matrices

**Solutions:**
- Pack/unpack functions
- Special (common) data types:
  - Array of data types
  - Array of memory displacements
- Managed by the message-passing library
**MPI:**

Data types for message contents:

- Standard types:
  - `MPI_INT`
  - `MPI_FLOAT`
  - `MPI_CHAR`
  - `MPI_DOUBLE`
  - ...

- User defined types: derived from standard types