Parallel programming
Message-passing

Salvatore Orlando
Principles of Message Passing Programming

- The logical view of a virtual machine supporting the message-passing paradigm (MP) consists of
  - $p$ processes/tasks, each with its own exclusive/private address space
  - It is the primitive programming model for architectures falling within the DM MIMD class

- Each data element must belong to one of the private partitions of the address space
  - data must be explicitly partitioned and placed in the private address spaces of each process

- All interactions (read-only or read/write) between processes require explicit cooperation of two processes
  - Between the process that owns the data and the process that wants to access the data

- This constraint, while onerous, make the underlying costs very explicit to the programmer
Principles of Message Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms
  - all concurrent tasks execute asynchronously.
  - tasks or subsets of tasks synchronize to perform interactions.
Message Passing Programming

• Languages or libraries?

• Languages
  – Send/Receive statements, specific constructs to define communication channels, etc. as an integral part of the language
  – OCCAM is an old MP language, whose MP constructs were directly supported by the Transputer machine language
  – Languages permit compile-time analyses, type checking, deadlock checking, etc.

• Libraries offer a set of MP primitives, and are linkable to many sequential languages (C/C++, F77/F90, etc.)
  – MPI (Message Passing Interface)
  – PVM (Parallel Virtual Machine)
  – Optimized version for specialized networks, but there exist versions that work for TCP/IP over an Ethernet network
MPI and PVM

- **MPI (Message Passing Interface)**
  - standard for parallel programming
  - Universities, Research centers, Industries were involved
  - there exist public-domain implementations
  - mpich – maintained by Argonne National Laboratories

- **PVM (Parallel Virtual Machine)**
  - first MP library that has been largely adopted
  - Homogeneous high-performance clusters, but also heterogeneous distributed architectures composed of remote hosts over Internet
  - developed by Oak Ridge National Laboratories
  - public domain
Basic mechanisms

• Process creation
  – At loading time
    • process number decided at loading time
    • used in MPI (mpich), now available on PVM too
    • SPMD (Single Program Multiple Data): same code executed by the all the process copies
  – At running time
    • given an executable code, create a process executing that code
    • used in PVM (spawn)
    • in principle, processes can execute different codes (MPMD – Multiple Programs Multiple Data)
  – At compiling time
    • old approach: all is decided statically (number and mapping of processes)
    • OCCAM su Meiko CS1 – Transputer
SPMD (Single Program Multiple Data)

Different compilations for the diverse platforms, if these machines are heterogeneous

Creation of $n$ copies of the same process

Loading on $n$ processors (real or virtual) with these processes, running the same code
Dynamic MPMD (Multiple Programa Multiple Data)

Processo 0

spawn()

Processo 1

EXE1

EXE2
Basic Mechanisms

- **Send/Receive of messages**
  - how are communication partners named?
  - group communication
    - collective operations
  - transmission of typed “data”
    - marshaling on heterogeneous machines (encoding/decoding data wrt a standard data serialization format)
  - transmission of typed “messages”
    - tags associated with messages
  - synchronous/asynchronous and blocking/non blocking
  - non deterministic reception of messages
Process naming

- In MP libraries processes are named by means of
  - numeric IDs (rank, tid, pid)

- SPMD (loading time process creation)
  - there are routines that allow the calling process to know
    - its own ID
    - the range of the contiguous process IDs (number of allocated processes)

- Dynamic creation
  - the routines that spawn processes return the IDs of the created processes, like fork()

- Non deterministic receiving
  - wild card in the receive(), to allow messages to be received from a set of possible senders
Synchronous communication protocol

- without system buffer (unbuffered), blocking

- `send()` cannot complete if the receiver is not arrived to execute the corresponding `receive()`

- realize a simultaneous communication and synchronization of both sender/receiver

- **Idling time** and **deadlocks** are the main issues of the synchronous communications
Synchronous communication protocol

Handshake for a blocking non-buffered send/receive operation.

It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.
Asynchronous/Buffered communication protocol

<table>
<thead>
<tr>
<th>Buffered</th>
<th>Non-Buffered</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking Operations</strong></td>
<td><strong>Non-Blocking Operations</strong></td>
</tr>
<tr>
<td>Sending process returns after data has been copied into communication buffer</td>
<td>Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return</td>
</tr>
<tr>
<td>Send and Receive semantics assured by corresponding operation</td>
<td>Programmer must explicitly ensure semantics by polling to verify completion</td>
</tr>
</tbody>
</table>
Asynchronous/Buffered communication protocol

- The `send()` primitive soon returns the control to the `sender` process
- The message is copied to the system buffer
  - the message is copied and transmitted asynchronously
  - it can arrive at the destination host, copied in a remote system buffer, even if `receiver` process has not yet invoked the corresponding `recv()`
- buffers have a fixed sized
  - the flow control can “block” the sender

![Diagram of Asynchronous/Buffered communication protocol]

The receive is blocking
Asynchronous/Buffered communication protocol

- Buffering
Blocking vs Non-Blocking

- In MPI, also the buffered/asynchronous routines are defined *blocking*
  - since they wait for copying the message to a system buffer

- Non-Blocking MPI routines
  - *return soon* even if their task has not yet been completed
    - the send() routine returns soon even if the message has not yet been copied to a buffer
    - the recv() returns soon even if the waited messages has not yet been arrived
  - the user memory, which either stores the message to be sent or will store the received message, *cannot be reused* till
    - the completion of the message send (copy from the user memory to a buffer of the communication subsystem)
    - the completion of the receive (copy from the system buffer to the user memory)
  - there exists MPI routines to check (*polling*) the completion of non-blocking routines

- The use of non-blocking routines permits programming styles able to overlap useful computation with message communication
Buffered and Unblocking protocols

Buffered:
- Blocking Operations:
  - Sending process returns after data has been copied into communication buffer
  - Send and Receive semantics assured by corresponding operation

Non-Buffered:
- Blocking Operations:
  - Sending process blocks until matching receive operation has been encountered

Non-Blocking Operations:
- Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
  - Programmer must explicitly ensure semantics by polling to verify completion
Non-buffered and Unblocking protocols

- In general, such protocols try to avoid message copies, since the copies introduce overheads due to the large memory latencies.

Message copies may also be avoided by using **blocking synchronous protocols (unbuffered)**
- low performance, due to the mutual waiting of sender/receiver

**Non-blocking primitives** return before completing
- we need to test later the completion of the delivery
Non-buffered and Non-Blocking MP

- Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.
- In case (b), the time for transmission and message copy is completely overlapped with useful computation on both communication sides.
MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.
### MPI: the Message Passing Interface

**The minimal set of MPI routines.**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI.</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI.</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes.</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of calling process.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message.</td>
</tr>
</tbody>
</table>
Starting and Terminating the MPI Library

- **MPI_Init** is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.

- **MPI_Finalize** is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.

- The prototypes of these two functions are:
  
  ```c
  int MPI_Init(int *argc, char ***argv)
  int MPI_Finalize()
  ```

- **MPI_Init** also strips off any MPI related command-line arguments.

- All MPI routines, data-types, and constants are prefixed by “MPI_”. The return code for successful completion is MPI_SUCCESS.
Hello World in MPI

```c
#include <mpi.h>
main(int argc, char *argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n", myrank, npes);
    MPI_Finalize();
}
```
Differentiate the behaviour of SPMD processes

- The processes can ask the MPI run-time for determining their ranks
  - they differentiate their behaviour on the basis of these ranks

```c
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
        master();
    else
        slave();
    MPI_Finalize();
}
```
Typing messages and transmitted data

- Tags
  - numerical identifiers to give a type to (to tag) a message
  - allow to distinguish between (semantically) different messages exchanged by a pair of processes
  - the matching between send and receive is also based on the tags

- Datatypes
  - the type of the transmitted data is needed for marshalling data when the communication occurs between heterogeneous hosts

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Standard MPI Send/Receive routines

### MPI_Send

- **buf**: Address of send buffer
- **count**: Number of items to send
- **datatype**: Datatype of each item
- **dest**: Rank of destination process
- **tag**: Message tag
- **comm**: Communicator

### MPI_Recv

- **buf**: Address of receive buffer
- **count**: Maximum number of items to receive
- **datatype**: Datatype of each item
- **src**: Rank of source process
- **tag**: Message tag
- **comm**: Communicator
- **status**: Status after operation

Asynchronous/buffered, blocking

Note the *communicator* (explained in the next slide)
Communicators

- A communicator defines a *communication domain* (a sort of *group identifier*)
  - a set of processes that are allowed to communicate with each other
  - the ranks allowed in a communicator including *n* processes are numbers between 0 and *n*-1
Communicators

- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators must be used as arguments of all the MPI routines:
  - Tag and process rank refer to a specific communicator.
  - Each process can participate in more communicators:
    - In principle, they can have distinct ranks for each communicator.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes created at loading time:
  - It is possible to create new communicators at run-time with specific collective routines, i.e., routines involving all the process group.
Other MPI communication modes

• Besides the Standard mode, MPI includes other communication modes and associated routines:
  – **Synchronous** (**MPI_Ssend**): the send does not complete till the corresponding receive is started (prone to **deadlocks**)
  – **Buffered** (**MPI_Bsend**): the user supplies the buffer to copy messages (Warning: the user has to supply enough buffer memory to avoid blocking programs for insufficient memory)
Wildcard, non determinism, etc.

- **Wildcards**
  - The receiver process can specify wildcards for both the source and the tag, in order to receive in a non deterministic way many messages, also arriving from several senders
    - MPI_ANY_SOURCE
    - MPI_ANY_TAG

- **Message length**
  - The count on the receiver must be large enough to receive the incoming message
    - must be greater than or equal to the length of the message to receive
  - After the reception, we can check the various features of the received message
Sending and Receiving Messages

- On the receiving end, the status variable can be used to get information about the MPI_Recv operation. The corresponding data structure contains:

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

- The MPI_Get_count function returns the precise count of data items received.

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

```c
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;

MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )

recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```
Querying Information

- The MPI_Comm_size and MPI_Comm_rank functions are used to determine the number of processes and the label of the calling process, respectively.
- The signatures of these routines are:
  
  ```c
  int MPI_Comm_size(MPI_Comm comm, int *size)
  int MPI_Comm_rank(MPI_Comm comm, int *rank)
  ```
- The rank of a process is an integer that ranges from zero up to the size-1, where size is the number of processes of communicator.
Non blocking routines

MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)

- The completion of both the routines can be tested by `MPI_Test()`
- `MPI_Test()` can be used to poll. It returns a flag that when equal to 1, means that the routine completed with success
- `MPI_Wait()` is instead the blocking waiting for the completion of the routine
Non blocking routines: an example

- In some implementations, the non-blocking routines allow computation and communication overheads to overlap
  - e.g., if the communication subsystem can directly access (DMA) user buffer (variable: \( x \)) while the user program is executing

```c
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

- Note the combined use of Non-Blocking Send and Standard blocking Recv
Unsafe programs and deadlocks

- Even using asynchronous primitives (buffered), we can generate deadlocks:
  - Suppose you need to send a message of large size from process 0 to process 1
  - If the system memory buffer is insufficient to store the message, the `send()` routine does not return, and waits for the corresponding `receive()` that supplies user space to store the message

- The following code is “unsafe”, since its correct behavior depends on the availability of system buffer, and can generate a deadlock

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>_recv (1)</td>
<td>_recv (0)</td>
</tr>
</tbody>
</table>
Solutions to the issue of “unsafe” code

- Order carefully the operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking primitives:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
Avoiding Deadlocks

Consider the following piece of code, in which each process $i$ sends a message to process $i + 1$ (module the number of processes) and receives a message from process $i - 1$ (module the number of processes)

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...

We have a deadlock if MPI_Send blocks
```
Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
Again Communicators

- The communicators allow a modular design of parallel library components
  - e.g.: we have an SPMD program where all the processes call the same library routine (which in turn is SPMD)
  - without communicators, it becomes important that the tags used in the library routine are distinct from the ones used by the rest of the application
  - by introducing a new communicator, the library routine can freely use arbitrary tags in the message exchange
Usage of communicators for parallel libraries

- SPMD program involving three processes, calling two library MPI subroutines
  - Sub1() and Sub2()
  - note: subroutines called collectively by all the processes (SPMD code)

- Correct behavior:
Usage of communicators for parallel libraries

- Using the same communicators, due to ANY_TAG, the behavior could be incorrect:
Collettive operations

- Collective operation
  - involves all the processes of a group (communicator)
  - must be invoked by all these processes

- An example of simple collective operation is the barrier synchronization:

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```
Broadcast/Multicast

- In MPI, `bcast()` is a collective operation

```c
int MPI_Bcast (void *buf, int count, 
               MPI_Datatype datatype, 
               int source, MPI_Comm comm)
```
Another collective operation is `gather()`

- A process collects all the disjoint parts of a data structure

```c
int MPI_Gather(void *sendbuf, int sendcount,
                MPI_Datatype senddatatype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvdatatype,
                int target, MPI_Comm comm)
```
Scatter

- Also gather() is collective
  - The opposite of gather()
Reduce

• **Not only communication** (in this case, a sort of *gather*), but also execution of an **associative operation** on data (e.g. product, sum, ecc.)
  – it could be implemented in an optimized manner, by avoiding the execution of all the computation by Process 0 (root)
  – in this regard, consider that, thanks to the associative property of the operation, in the implementation some partial reductions can be executed by the various processes involved, before producing the final reduction

```c
int MPI_Reduce(void *sendbuf, void *recvbuf,
                int count, MPI_Datatype datatype,
                MPI_Op op, int target, MPI_Comm comm)
```
### Predefined Reductions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_MAX</code></td>
<td>Maximum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td><code>MPI_MIN</code></td>
<td>Minimum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td><code>MPI_SUM</code></td>
<td>Sum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td><code>MPI_PROD</code></td>
<td>Product</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td><code>MPI_LAND</code></td>
<td>Logical AND</td>
<td>C integers</td>
</tr>
<tr>
<td><code>MPI_BAND</code></td>
<td>Bit-wise AND</td>
<td>C integers and byte</td>
</tr>
<tr>
<td><code>MPI_LOR</code></td>
<td>Logical OR</td>
<td>C integers</td>
</tr>
<tr>
<td><code>MPI_BOR</code></td>
<td>Bit-wise OR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td><code>MPI_LXOR</code></td>
<td>Logical XOR</td>
<td>C integers</td>
</tr>
<tr>
<td><code>MPI_BXOR</code></td>
<td>Bit-wise XOR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td><code>MPI_MAXLOC</code></td>
<td>max-min value-location</td>
<td>Data-pairs</td>
</tr>
<tr>
<td><code>MPI_MINLOC</code></td>
<td>min-min value-location</td>
<td>Data-pairs</td>
</tr>
</tbody>
</table>
Special reductions: `MPI_MAXLOC` and `MPI_MINLOC`

- The `MPI_MAXLOC` operation combines pairs of values \((v_i, l_i)\), and finally returns the pair \((v, l)\), such that
  - \(v\) is the maximum among all \(v_i\)
  - \(l\) is the \(l_i\) corresponding to \(v_i\)
  - If more \(v_i\) are equal to the maximum value, the returned \(l\) is the smallest of all \(l_i\)
- `MPI_MINLOC` behaves analogously, but returns the minimum value \(v_i\) rather than the maximum one

<table>
<thead>
<tr>
<th>Value</th>
<th>15</th>
<th>17</th>
<th>11</th>
<th>12</th>
<th>17</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
\text{MinLoc}(\text{Value, Process}) = (11, 2) \\
\text{MaxLoc}(\text{Value, Process}) = (17, 1)
\]
Collective operations All_*

- When all the processes must receive the same result of a collective operation

```c
int MPI_Allreduce(void *sendbuf, void *recvbuf,
                   int count, MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm)
```

```c
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI_Datatype recvdatatype,
                  MPI_Comm comm)
```

- All-to-all personalized communication
  - send a distinct message from every process to each other process
    - $n^2(n-1)$ distinct messages
  - the $j$-th block of output data ($sendbuf$), sent from task $i$, is received from task $j$
  - it is put in the $i$-th block of the input buffer ($recvbuf$)

```c
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, void *recvbuf,
                 int recvcount, MPI_Datatype recvdatatype,
                 MPI_Comm comm)
```
Topologies and Embedding

• MPI allows a programmer to organize processors into logical $n$-dimension meshes

• The processor ids in MPI_COMM_WORLD (1-d mesh) are mapped to other communicators (corresponding to higher-dimensional meshes) in many ways

• The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machine

• MPI often does not provide the programmer any control over these mappings
  – anyway, some MPI programming environments for MMP (IBM BlueGene) allows to control the mapping
### Topologies and Embedding

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>15</td>
<td>12</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **(a) Row-major mapping**
- **(b) Column-major mapping**
- **(c) Space-filling curve mapping**
- **(d) Hypercube mapping**

**Figure 6.5** Different ways to map a set of processes to a two-dimensional grid. (a) and (b) show a row- and column-wise mapping of these processes, (c) shows a mapping that follows a space-filling curve (dotted line), and (d) shows a mapping in which neighboring processes are directly connected in a hypercube.
Creating and Using Cartesian Topologies

• We can create cartesian topologies using the function:

```c
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                     int *dims, int *periods, int reorder,
                     MPI_Comm *comm_cart)
```

This function takes the processes in the old communicator and creates a new communicator with \textit{ndims} dimensions.

• Each processor can now be identified in this new Cartesian topology by a vector of dimension \textit{ndims}

• The max size for each dimension is given by:
  – \textit{dims[i]} \rightarrow a different \textit{k} for each dimension of the \textit{k-ary n-cube}

• If \textit{periods [i]} \neq 0, then the topology has a wraparound connection on the \textit{i-th} dimension

• If \textit{reorder=True}, the function should reorder the processes to choose a good embedding of the virtual topology on the physical one
  – this parameter is very often ignored by the library implementation
Creating and Using Cartesian Topologies

• Since sending and receiving messages still require (one-dimensional) ranks, MPI provides routines to convert ranks to Cartesian coordinates and vice-versa:

  Rank ➔ Cartesian Coordinates
  \[
  \text{int } \text{MPI\	extunderscore Cart\	extunderscore coords} (\text{MPI\	extunderscore Comm } \text{comm\	extunderscore cart}, \text{int } \text{rank}, \text{int } \text{maxdims}, \\
  \text{int } *\text{coords})
  \]

  Cartesian Coordinates ➔ Rank
  \[
  \text{int } \text{MPI\	extunderscore Cart\	extunderscore rank} (\text{MPI\	extunderscore Comm } \text{comm\	extunderscore cart}, \text{int } *\text{coords}, \text{int } *\text{rank})
  \]

• The most common operation on the Cartesian topologies is a shift. To determine the rank of source and destination of such shifts, MPI provides the following function:

  \[
  \text{int } \text{MPI\	extunderscore Cart\	extunderscore shift} (\text{MPI\	extunderscore Comm } \text{comm\	extunderscore cart}, \text{int } \text{dir}, \text{int } s\text{\_step}, \\
  \text{int } *\text{rank\	extunderscore source}, \text{int } *\text{rank\	extunderscore dest})
  \]

  where:
  – \text{dir} is the dimension along which we want to perform the shift
  – \text{s\_step} indicates how many shift steps must be performed
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.

- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.

- The simplest such mechanism is:
  
  ```c
  int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
  ```

- This operation groups processors by color and sorts resulting groups on the key.
Groups and Communicators

Using MPI_Comm_split to split a group of processes in a communicator into subgroups
MPI-2

- One sided communication (remote memory access)
  - Processes can access another process address space without any explicit participation in that communication operation by the remote process
  - MPI Get - remote read
  - MPI Put - remote write
  - MPI Accumulate - remote update, reduction
  - Communication and synchronization are uncoupled

• Dynamic Creation / Termination of processes
• Parallel I/O
MPICH

- MPICH is a portable MPI implementation (public domain)
- Works on MPPs, Grids, clusters (both homogeneous and heterogeneous)
- To compile
  ```
  mpicc -o myprog myprog.c
  ```
MPICH (compilation)

- There exists different scripts to compile and link (include paths and libraries to pass to the sequential compiler are specified by the script)
  - C (mpicc)
  - C++ (mpiCC)
  - Fortran 77 (mpif77)
  - Fortran 90

- The compilation scripts allow several options:
  - e.g.: -mpilog
    - the executable Costruisce eseguibile che genera MPE log files.

- Possibile linkare altre librerie
  - mpicc -o foo foo.o -lm
MPIEXEC

• Running MPICH2 application programs:
  – MPI uses Hydra as process management system for starting parallel jobs
• Hydra is designed to natively work with multiple launcher on remote/local hosts
  – A launcher is the basic remote node access mechanism that is provided on any system. Hydra supports multiple launchers including ssh, rsh, fork, and slurm to launch processes. All of these are compiled in by default, so you can pick any one of them at runtime using the mpiexec option –bootstrap
  – The default is ssh

• mpiexec starts many instances of the same program (SPMD) using Hydra
  mpiexec -n 5 ./hello
MPIEXEC

- Create a file with the names of the machines that you want to run your job on. This file may or may not include the local machine
  
  ```shell
  $ cat hosts
  donner
  foo
  shakey
  terra
  ```

- To run your application on these nodes, use mpiexec:
  
  ```shell
  $ mpiexec -f hosts -n 4 ./hello
  ```

- If the same application is run with 8 processes, it will use the hosts in a round robin manner.
  - For instance, with the above host file, ranks 0 and 4 will execute on donner, ranks 1 and 5 on foo, ranks 2 and 6 on shakey and ranks 3 and 7 on terra.
• The host file can also be specified as follows:

```
shell$ cat hosts

# This is a sample host file
donner:2     # The 1st 2 procs are scheduled to run here
foo:3        # The next 3 procs run on this host
shakey:2     # The last 2 procs run on this host
```

• mpiexec can also launch an MPMD program:

```
mpiexec -n 1 master : -n 19 slave
```
rsh/ssh

- To launch remote processes, we have to reach these machines with ssh or rsh without entering a password
  - Single sign-on

- rsh
  - This mechanism is not secure
  - A host allows remote users to login if they are included as user@host in a file (.rhosts)
  - The login is allowed only on the basis of an IP address and a user name

- ssh
  - It is more secure, since it is based on a public key cryptography
  - The remote host, which we need to login to, has to know the public key of the client user@host
  - The client shows his/her identity to the remote host by signing (with the corresponding private key) a challenge message, which s/he get when connecting the remote host
  - The host can verify the identity since it know the public key
  - The security is thus guaranteed by protecting careful the private key

Probably Kerberos also guarantees single sign-on, but for a short temporal interval after the sign-on
SSH

- Generation of public/private keys:
  - `ssh-keygen -t rsa`
    - for each password request, simply enter <Invio>
    - however, in this way the private key shall not be protected by any pin
  - The successfully execution of this command generates in `~/ssh` the files:
    - `id_rsa` (private key)
    - `id_rsa.pub` (public key)

- Copy the file `~/ssh/id_rsa.pub` in `~/ssh/authorized_keys`
  - Since in our lab the home directory is shared (is mounted from the file server), it is enough a single copy to guarantee the one-time password access from any host in the lab
  - You can now execute the same test suggested for `rsh`

- Hint:
  - Put the key `~/ssh/id_rsa` in a safe memory support
  - You can remove the key when logout
Hello World

• In the following program `hello.c` all the processes print a string that reports the MPI rank of each process

• The strings are collected by the task whose rank = 0
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]){
    int my_rank; /* rank of process */
    int p;  /* number of processes */
    int source;  /* rank of sender */
    int dest;  /* rank of receiver */
    int tag=0;  /* tag for messages */
    char message[100];  /* storage for message */
    MPI_Status status;  /* return status for receive */
    /* start up MPI */
    MPI_Init(&argc, &argv);

    /* find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

Hello World

```c
if (my_rank != 0) {
    /* create message */
    sprintf(message, "Hello MPI Pi World from process %d!", my_rank);
    dest = 0;
    /* use strlen+1 so that '\0' get transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else {
    printf("Hello MPI Pi World From process 0: Num processes: %d\n", p);
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("%s\n", message);
    }
}
/* shut down MPI */
MPI_Finalize();
return 0;
```
• **cpi** is a classic **SPMD** program example (see the following slides) which computes the value of $\pi$ by numerical integration in parallel.
  
  ```
  mpiexec -n 5 cpi
  ```

• The **cpi** example will tell you which hosts it is running on.
The area of a circle is $r^2 \pi$. So the area of a quarter of a circle (quadrant) for $r=1$ is: 

\[ \frac{\pi}{4} \]

Equation of a circle with the center in $(0,0)$:

\[ x^2 + y^2 = 1 \]

\[ y = \sqrt{1-x^2} \]

We can compute the area of the quadrant by the following integral:

\[ \int_{0}^{1} \sqrt{1-x^2} \, dx \]

Equivalently we can compute

\[ \int_{0}^{1} \frac{1}{1+x^2} \, dx \]

ARCTAN(1) = $\pi/4$

ARCTAN(0) = 0

We can compute the integral numerically

The numerical precision improves when we increases the number of intervals of the segment $[0..1]$
CPI: computing $\pi$ with a parallel MPI C program (1)

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

double f(double a) {
    return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);
```
CPI: computing $\pi$ with a parallel MPI C program (1)

```c
fprintf(stdout, "Process %d of %d on %s\n", myid, numprocs, processor_name);
fflush(stdout);

n = 0;
while (!done) {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        fflush(stdout);
        scanf("%d", &n);

        startwtime = MPI_Wtime();
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (n == 0)
        done = 1;
    else {
        h   = 1.0 / (double) n;
        sum = 0.0;
```
CPI: computing $\pi$ with a parallel MPI C program (2)

for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += f(x);
}

mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

the median point of the i-th segment of size h
CPI: computing π with a parallel MPI C program (2)

if (myid == 0) {
    printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    endwtime = MPI_Wtime();
    printf("wall clock time = %f\n", endwtime - startwtime);
    fflush( stdout );
}
}

MPI_Finalize();
return 0;
Some simple exercises

- **Modify** `cpi` in order to exploit simple point-to-point send/receive rather than `bcast/reduce`, and to receive the input as a command line argument.

- **Write a program that forward a message token along a ring:**
  - Process 0 reads a line from `stdin`
  - Process 0 send the read line to Process 1, which forwards to Process 2, and so on.
  - The last process of the ring sends back to Process 0, which prints the line.
  - **Use** `MPI_Wtime` **to measure the execution time** (`man MPI_Wtime`)

- **Compute the transmission bandwidth and transmission overhead between a pair of processes**
  - **Ping-pong:**
    - the former sends and then receives, the latter receives and then sends
  - **Measure with** `MPI_Wtime` **the time taken on the former process**, and returns the time divided by the number of transmitted Byte
  - What does it happen when we increase the message size?
Example of master-slave

• Divide the processors into two communicators
  – one including only the master
  – one including the slaves

• The master accepts messages from the slaves (of type MPI_CHAR) and print them in rank order (that is, first from slave 0, then from slave 1, etc.)
  – The ranks of the slaves is related to the new communicator

• Each slave send 2 messages to the master.
  – “Hello from slave <n>”
  – “Goodbye from slave <n>”

• We have different TAG’s used to type messages (hello or goodbye)
Example of master-slave

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

#define HELLO_TAG 0
#define GOODBYE_TAG 1

void master_io(MPI_Comm master_comm, MPI_Comm comm);
void slave_io(MPI_Comm master_comm, MPI_Comm comm);

.....
Example of master-slave

```c
int main(int argc, char **argv)
{
    int rank;
    MPI_Comm new_comm;
    
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_split( MPI_COMM_WORLD, rank == 0, 0, &new_comm );
    
    if (rank == 0)
        master_io( MPI_COMM_WORLD, new_comm );
    else
        slave_io( MPI_COMM_WORLD, new_comm );
    
    MPI_Finalize( );
    return 0;
}
```
Example of master-slave

/* This is the master */
void master_io(MPI_Comm master_comm, MPI_Comm comm)
{
    int i, size;
    char buf[256];
    MPI_Status status;

    MPI_Comm_size(master_comm, &size);
    for (i=1; i<size; i++) {
        MPI_Recv(buf, 256, MPI_CHAR, i, HELLO_TAG,
                  master_comm, &status);
        fputs(buf, stdout);
    }

    for (i=1; i<size; i++) {
        MPI_Recv(buf, 256, MPI_CHAR, i, GOODBYE_TAG,
                  master_comm, &status);
        fputs(buf, stdout);
    }
}
Example of master-slave

/* This is the slave */
void slave_io(MPI_Comm master_comm, MPI_Comm comm)
{
    char buf[256];
    int rank;

    MPI_Comm_rank( comm, &rank );

    sprintf( buf, "Hello from slave %d\n", rank );
    MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0,
              HELLO_TAG, master_comm);

    sprintf( buf, "Goodbye from slave %d\n", rank );
    MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0,
              GOODBYE_TAG, master_comm );
}
PVM: an older message passing library

- The PVM (Parallel Virtual Machine) project was started in 1989 by the Oak Ridge National Laboratory
  - Prototype (not released), PVM 1.0, designed by Vaidy Sunderam and Al Geist

- The version 2 of PVM was written by the University of Tennessee (Dongarra et al.)
  - Public domain version released in 1991

- Current version:
  - Both Windows and Unix
  - New features: communication contexts, persistent messages (fault tolerance)
PVM

- At the beginning, we create the PVM by launching a daemon on each host
  - by using a console it is possible to interact with the daemons, and manage the concurrent tasks
PVM architecture

**pvmd** - one PVM daemon per host

**libpvm** - task linked to PVM library

---

The collection of communicating **PVMDs** define the Parallel Virtual Machine
Step 1 – Enrolling in PVM

- Tasks must first be enrolled in the PVM
  
  ```c
  myid = pvm_mytid();
  printf(" \-> \%x", myid);
  \-> 0x40001
  ```

- The first call to a PVM routine enrolls the task in the PVM
  - The usual way is to call `pvm_mytid()` as the first invoked PVM routine
Sending/receiving messages

- **Send a message**
  
  ```c
  err = pvm_send( dest, tag );
  Dest = TID of destination;
  Tag = user-defined distinguishing integer;
  ```

- **Receive a message**
  
  ```c
  Err = pvm_recv( src, tag );
  Src = TID of sender
  Tag = distinguished integer
  ```
Wildcards

- Wildcards are possible in the `recv()` for the tid of sender, for the matching tag

- The wildcard is -1

  *Any sender, specific tag*
  
  ```
  pvm_recv(-1, tag);
  ```

  *Specific sender, any message*
  
  ```
  pvm_recv(src, -1);
  ```

  *Any sender, any message*
  
  ```
  pvm_recv(-1, -1);
  ```
PVM Console

• If you issue the command `pvm`, you are asking to launch locally the PVM console
  – If there are no active VMs for the user, the command also starts a new VM
    • A daemon pvmd is started on the local host
    • The console is enrolled in the VM and waits for the user commands
  – If there are already no active VMs, the console is simply enrolled in the existing VM and waits for the user commands

• The PVM console is exactly a new PVM task
  – All the stuff that can be done by using the console, can also be done with suitable APIs
## PVM Console

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add <code>hostname</code></td>
<td>Add host(s) to virtual machine (can list several)</td>
</tr>
<tr>
<td>conf</td>
<td>List hosts in the virtual machine</td>
</tr>
<tr>
<td>delete <code>hostname</code></td>
<td>Delete hosts from virtual machine</td>
</tr>
<tr>
<td>halt</td>
<td>Shut down PVM and tasks</td>
</tr>
<tr>
<td>help <code>[command]</code></td>
<td>Print information about commands and options</td>
</tr>
<tr>
<td>kill <code>tid</code></td>
<td>Kill a task</td>
</tr>
<tr>
<td>ps <code>-a</code></td>
<td>List all running tasks</td>
</tr>
<tr>
<td>quit</td>
<td>Exit console - leave PVM and tasks running</td>
</tr>
<tr>
<td>reset</td>
<td>Kill all tasks and reset PVM</td>
</tr>
<tr>
<td>spawn</td>
<td>Spawn tasks (many options)</td>
</tr>
<tr>
<td>trace</td>
<td>Set/display trace events</td>
</tr>
<tr>
<td>version</td>
<td>Print PVM version</td>
</tr>
</tbody>
</table>