Parallel Computing Paradigms

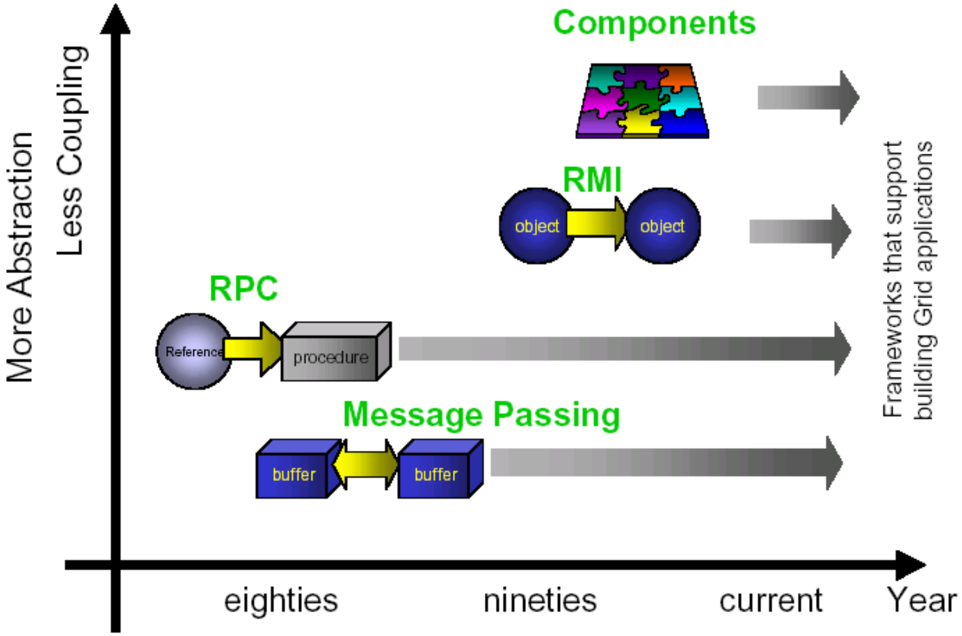
Message Passing

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

20 October 2015

Message Passing

Communication paradigms for distributed memory

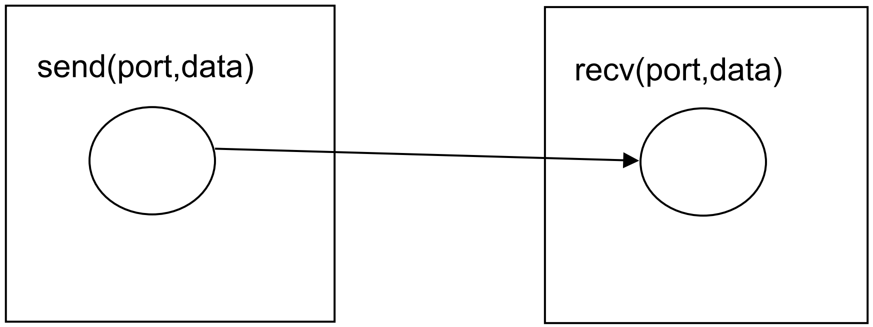


- Message passing is (still) more efficient than remote method invocations (RMI)

Message Passing

Basic concepts

- Specification of parallel activities through processes with disjoint address spaces
  - Processes can be identical (Single Process Multiple data, SPMD, e.g., MPI) or not (Multiple Instructions Multiple Data, MIMD, e.g., PVM)
- Activities can communicate/identified through ports
- Message send and reception is implicit (from/to a port)
- Data must be explicitly marshalled into messages
- There are more sophisticated communication primitives (broadcast, reduction, barrier)



Message Passing

Message Passing versus Remote method invocation (RMI)

	Message Passing	Remote Method Invocation
Data to send	Data packing into messages	List of parameters
Request for an action	Explicit send of tags in messages	Invocation of a specific method
Request reception	Explicit message reception	Implicit invocation of the requested method
Receptor reaction	Explicit programed as a funtion of the tag	Action is implicit in the invoked mothod
Receptor indentification	Chanel, name or anonimous	Pointer to remote object (proxy)

Message Passing

MPI (Message Passing Interface) http://www.mpi-forum.org

- **Standard** for message passing, outcome of an effort to provide a way to develop **portable** parallel applications
- Based on the SPMD model (the same process is executed on all machines)
  - The same executable is launched on a given set of machines
  - Each process has a unique identifier
  - In order message delivery
- Implemented as a library of functions
- Common Libraries (Open Source): OpenMPI, MPICH, e LamMPI
- Main features:
  - Several modes of message passing
    - Synchronous / asynchronous
  - Communication groups / topologies
  - Large set of collective operations
    - Broadcast, Scatter/gather, reduce, all-to-all, barrier
  - MPI-2
    - Dynamic processes, parallel I/O, Remote memory access, RMA (put/get)

Message Passing

Structure of a MPI program

- **Initialize the library**
  - **MPI_Init**
    - Initializes the library
  - **MPI_Comm_size**
    - Gets total number of process
  - **MPI_Comm_rank**
    - Get the id of current process
- **Execute the body of the program**
  - **MPI_Send**
  - **MPI_Recv**
- **Terminate**
  - **MPI_Finalize**

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char *argv[]) {
    int rank, msg;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        msg = 123456;
        MPI_Send( &msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv( &msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
        printf( “Received %d\n”, msg);
    }

    MPI_Finalize();
    return 0;
}
```

- Compile and execute the program
  - **compile:** mpicc orm picxx
  - **execute:** mpirun –np <*number of processes*> a.out

Message Passing

MPI – C++

- **Exemple 2 (C++)**

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[]) {
        int rank, buf;
        MPI::Init(argv, argc);
        rank = MPI::COMM_WORLD.Get_rank();
        // Process 0 sends and Process 1 receives
        if (rank == 0) {
                buf = 123456;
                MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
        }
        else if (rank == 1) {
                MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
                std::cout << "Received " << buf << "\n";
        }
        MPI::Finalize();
        return 0;
}
```

Message Passing

MPI (Functionalities – cont.)

- **Point to point communication between processes**

```
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)
```

- **It is required to provide the message data type (MPI_INT, MPI_DOUBLE, etc)**

- **Each process is identified by its rank in the group**
  - By default there is a group comprising all processes: MPI_COMM_WORLD
  - *dest* / *source* provide the destination / source of the message

- **The tag can be used to make distinction among messages**

- **MPI_Send:** will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).

- **reception:** waits for the arrival of a message with the required characteristics
  - MPI_ANY_SOURCE and MPI_ANY_TAG can be used to identify any source / any tag

Message passing

MPI – Modes of point-to-point communication

- **Message passing overhead**
  - Message transfer time (copy into the network, network transmission, deliver at the receptor buffer)
  - Wait for the receiving process

- **MPI_Ssend (blocking synchronous send)**
  - the sender waits until the message is received (w/ MPI_Recv)

- **MPI_Bsend (Buffered send)**
  - Returns as soon as the message has been placed on a buffer on the sender side
  - **Does not suffers from the overhead of receptor synchronization, but may copy to a local buffer**

- **MPI_Rsend (Ready send)**
  - Returns as soon as the message has been placed in the network
  - **The receptor side should already posted to avoid recv "deadlocks"**

- **MPI_Ixxx (non-bloking sends) c/ MPI_wait / MPI_Test /MPI_Probe**
  - Return immediately, being the programmer responsible to verify if the operation has completed (using wait)

Message Passing

MPI – Collective communications

- **int MPI_Barrier(MPI_Comm comm)**
  - Wait until all processes arrive at the barrier

- **int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)**
  - Broadcast the data from root to all other processes

- **int MPI_Gather e int MPI_Scatter**

(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)
  - **Gather:** Joints data from all processes into the root
  - **Scather**: scatters data from root into all other processes
  - **int MPI_Allgather** & **int MPI_Alltoall**

- **int MPI_Reduce**

(void* sbuf, void* rbuf, int count, MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm)
  - Combines the results from all process into the root, suing the operator *MPI_Op*
  - **int MPI_Allreduce** & **int MPI_Reduce_scatter**

Message Passing

Java implementations

- mpiJava – most well known MPI binding using JNI (requires MPI)
- MPJExpress – sucessor of mpiJava with support for MPI and “nio”;
- MPP – based on “nio” of Java 4; efficient version, does not follow the standard

```
    Communicator comm=new BlockCommunicator();   // package mpp.*
    int proc = comm.size();
    int rank = comm.rank();
     int [] buf = new int[1];
    // Process 0 sends and Process 1 receives
    if (rank == 0) {
        buf[0] = 123456;
        comm.send( buf, 0, 1, 1 );  // send(int[] data, [int offset, int length,] int peer)
    } else if (rank == 1) {
        comm.recv( buf, 0, 1, 0 );   // recv(int[] data, [int offset, int length,] int peer)
        System.out.println(“Recebi” + buf[0]);
    }
}
```