Master Informatics Eng.

2021/22

A.J.Proença

Programming with message passing

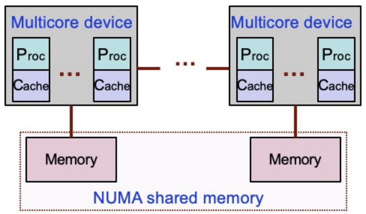
(most slides are from previous year)

Rewinding…

Current homogeneous parallel systems *(1)*

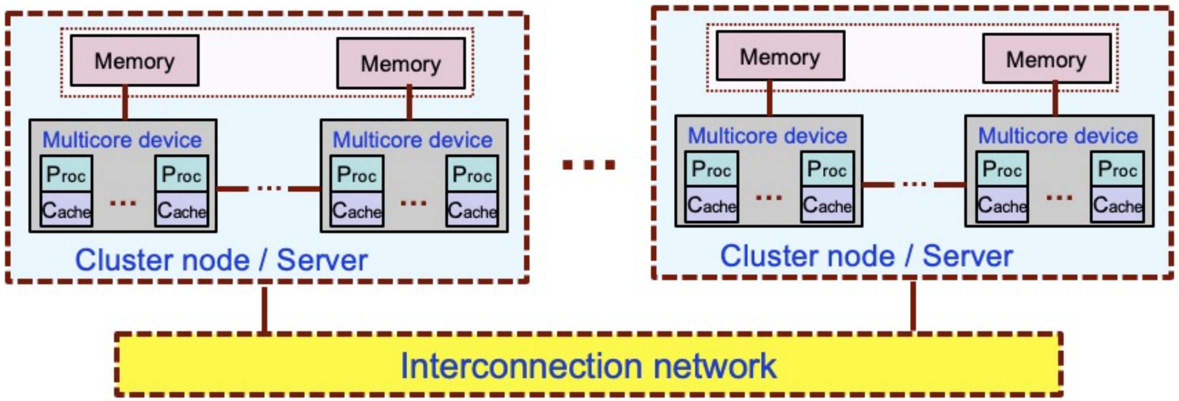
– parallelism on single or multiple devices *(same motherboard)*

- each core can be multithreaded
- single physical mem addr space
- paradigm: shared mem program
  - Cilk Plus (http://www.cilkplus.org/) extension to C & C++ to support data & task parallelism
  - **OpenMP** (http://openmp.org/wp/) C/C++ and Fortran directive-based parallelism



Explicit parallel computing *(2)*

- Current homogeneous parallel systems *(2)*
  – on multiple boards *(or multiple nodes/servers)*
    - each node with its private memory space
    - paradigm among nodes: distributed memory passing
      - MPI (https://en.wikipedia.org/wiki/Message_Passing_Interface) library for message communication on scalable parallel systems



Rewinding and advancing…

Parallel Programming Models

- Two general models of parallel program
  - Task parallel
    - problem is broken down into tasks to be performed
    - individual tasks are created and communicate to coordinate operations
  - Data parallel
    - problem is viewed as operations of parallel data
    - data distributed across processes and computed locally
- Characteristics of scalable parallel programs
  - Data domain decomposition to improve data locality
  - Communication and latency do not grow significantly

Shared Memory Parallel Programming

- Shared memory address space
- (Typically) easier to program
  - Implicit communication via (shared) data
  - Explicit synchronization to access data
- Programming methodology
  - Manual
    - multi-threading using standard thread libraries
  - Automatic
    - parallelizing compilers
    - OpenMP parallelism directives
  - Explicit threading (e.g. POSIX threads)

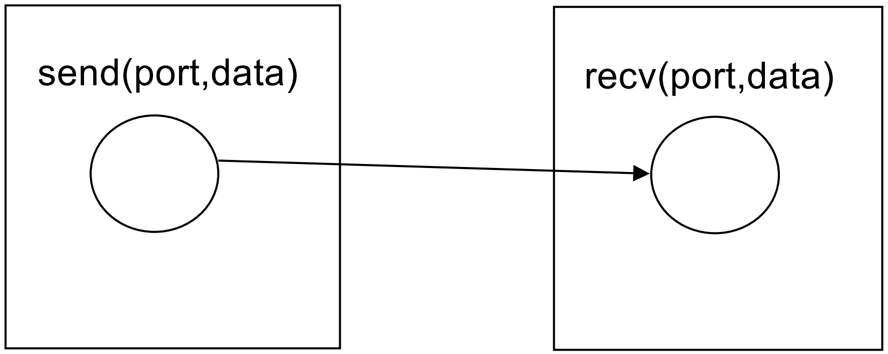
Distributed Memory Parallel Programming

- Distributed memory address space
- (Relatively) harder to program
  - Explicit data distribution
  - Explicit communication via messages
  - Explicit synchronization via messages
- Programming methodology
  - Message passing
    - plenty of libraries to chose from (MPI dominates)
    - send-receive, one-sided, active messages
  - Data parallelism

Message Passing

Basic concepts

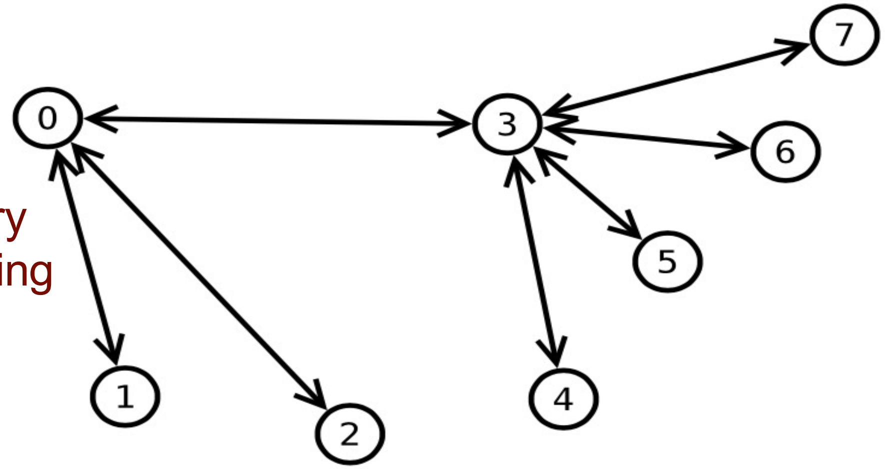
- Specification of parallel activities through **processes** with **disjoint address spaces**
  - No shared memory among processes => message passing parallelism
  - Processes can be identical (Single Program Multiple Data, SPMD) or not (Multiple Instructions Multiple Data, MIMD)
- Parallel activities **communicate** through ports or channels
  - Message **send** and **receive is explicit** (from/to a port or channel)

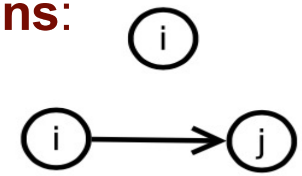


- **Data** must be **explicitly assembled** into messages
- There are more sophisticated communication primitives (broadcast, reduction, barrier)

A standard MPI application:

- **processes** that perform **computations** and interact with each other by explicitly **sending** and **receiving messages**
- **each process** is identified with a so-called **rank**, unique within a group of processes in an MPI communicator
- MPI spec supports **multithreading** within MPI processes
  - MPI-1: no shared memory concept
  - MPI-2: a limited distributed shared memory
  - MPI-3: explicit shared memory programming
  - MPI-4: current version (June 2021)
- **communication modes**:
  - point-to-point
  - collective / group
- **synchronization of communications**:
  - blocking (synchronous)
  - non-blocking (asynchronous)





MPI process with rank i

message sent from process with rank i to process with rank j

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 distributed memory)
- **Standard** to aid to develop **portable** parallel applications (on distributed memory)
- Useful on the **SPMD** model (where the same code is executed on all PUs)
- Message passing with **in-order message delivery** on point-to-point communication
- Implemented as a **library** of functions
- Common Libraries (<u>Open Source</u>): OpenMPI, MPICH and 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: with 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
- **Get information for process**
  - **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**
    - Do processing and send/recv data
- **And cleanup**
  - **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 & execute the program

- **compile:** `mpicc` (or `mpicxx` for C++)
- **execute:** `mpirun -np` *`<number of processes>`* `a.out`

Message Passing

MPI functionalities

- **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)
```

- **Message data content:** `void *buf, int count, MPI_Datatype datatype`
  - Requires the specification of the data type (`MPI_INT, MPI_DOUBLE, MPI_CHAR, …`)
- **Each process is identified by its rank in the group**
  - *dest* / *source* provides the destination / source of the message
  - By default there is a group comprising all processes: `MPI_COMM_WORLD`
- **The tag can be used to make a distinction among messages from the same rank**
- **`MPI_Recv`:** waits for the arrival of a message with the required characteristics
  - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be used to receive from any source / any tag

MPI Basic (Blocking) Send

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

- The message buffer is described by (`buf, count, datatype`).
- The target process is specified by `dest` and `comm`.
  - `dest` is the rank of the target process in the communicator specified by `comm`.
- `tag` is a user-defined "type" for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Basic (Blocking) Receive

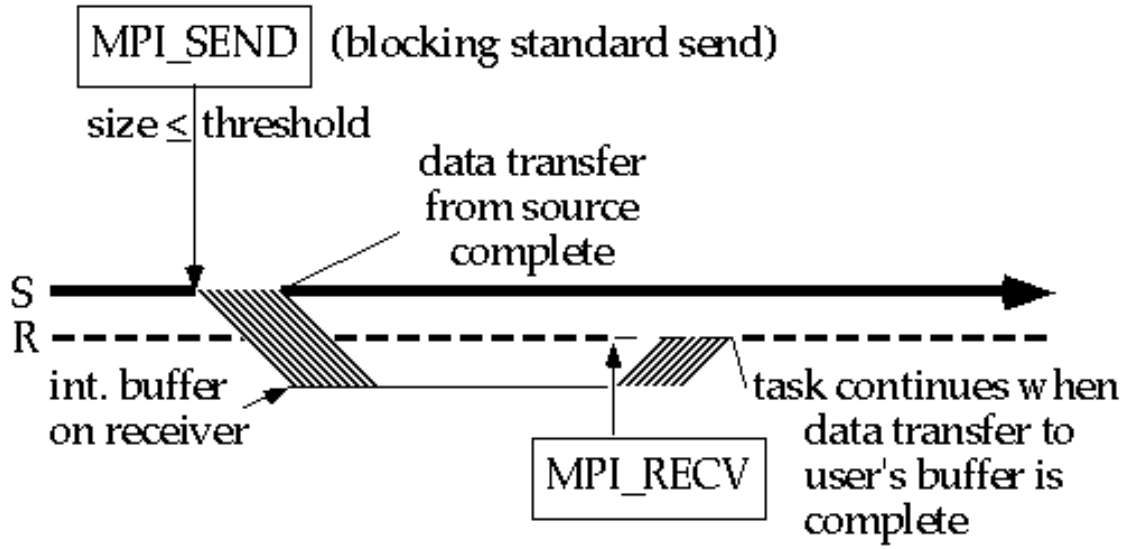
```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

- Waits until a matching (on `source`, `tag`, `comm`) message is received from the system, and the buffer can be used.
- `source` is rank in communicator `comm`, or `MPI_ANY_SOURCE`.
- Receiving fewer than `count` occurrences of `datatype` is OK, but receiving more is an error.
- `status` contains further information:
  - Who sent the message (can be used if you used `MPI_ANY_SOURCE`)
  - How much data was actually received
  - What tag was used with the message (can be used if you used `MPI_ANY_TAG`)
  - `MPI_STATUS_IGNORE` can be used if we don't need any additional information

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)



Message Passing

MPI – Modes of point-to-point communication

- **"Standard" MPI_Send may be implemented on a variety of ways**
  - "`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)" – *MPI standard*

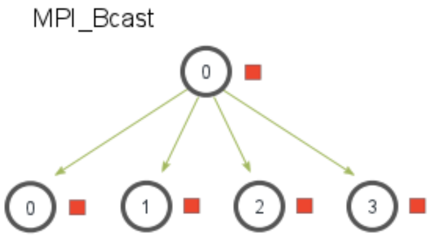
- **Explicit send implementations (**different options for **buffering** and **synchronization):**
  - **`MPI_Ssend`** (blocking Synchronous send)
    - The sender waits until the message is received (w/ `MPI_Recv` on the destination process)
  - **`MPI_Rsend`** (Ready send)
    - Returns as soon as the message has been placed in the network
    - The receptor side should already posted a `MPI_Recv` to avoid "deadlocks"
  - **`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

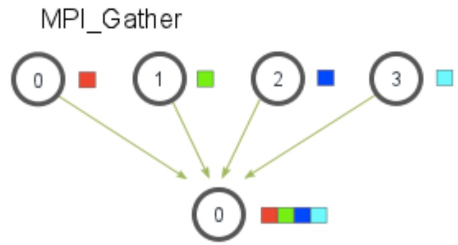
- **`MPI_Ixxx`** (non-blocking send) with **`MPI_wait`** / **`MPI_Test`** / **`MPI_Probe`**
  - Returns immediately; the programmer must verify if the operation has completed (using wait)

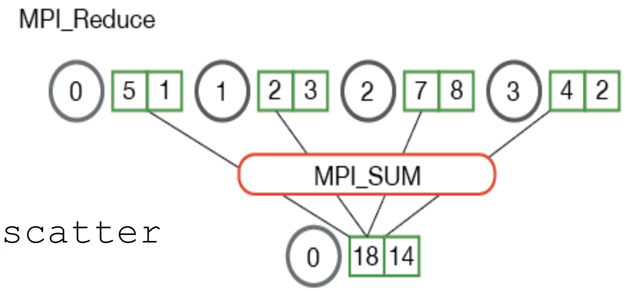
Message Passing

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 & 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_Reduce (void* sbuf, void* rbuf, int count, MPI_Datatype stype, MPI_O op, int root, MPI_Comm comm)`
  - **Combines** the results from all process into the root, using the operator `MPI_Op`
- Compositions: `Allgather, Alltoall , Allreduce, Reduce_scatter`



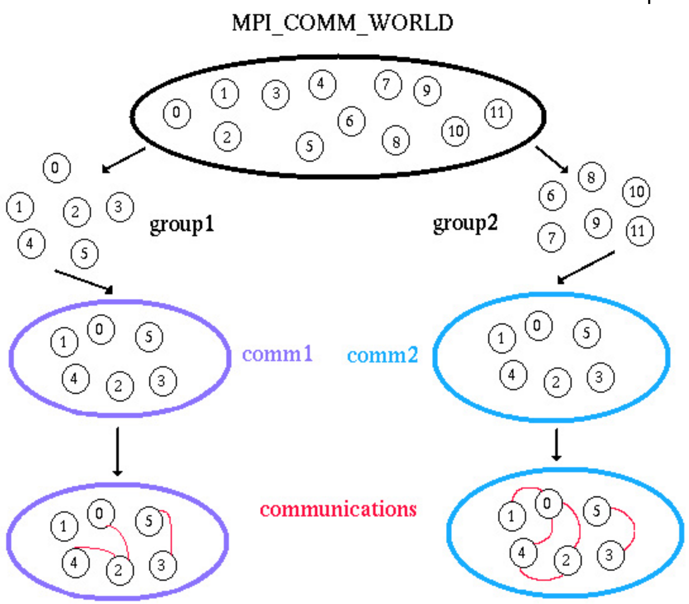




Message Passing

MPI – Groups

- Ordered group of process
  - Each process has a rank within the group
- Scope for communication on collective and point to point communications



MPI – Topologies

- Well defined structure of processes
  - Each process has a set of neighbours
    - Easier to identify with a topology
  - Communications though “channels”
  - Example: cartesian 4x4

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Performance of parallel applications

Measuring execution time

- **Execution time**
  - Time measured since the first **process** (or thread) starts execution until the last **process** (or thread) terminates (wall time)

    $$T_{exec} = T_{comp} + T_{comm} + T_{free}$$

    - T_{comp} : **computation time,** time spent in computations
      - Excludes communication/synchronization and free time
      - The sequential version can be used to estimate T_{comp}
    - T_{comm} : **communication time,** time spent sending/receiving messages
      - For each msg: the communication setup (t_s, includes msg build & latency), the communication bandwidth (=1/t_b ; throughput: effective b/w of an app) and the message length (L, in bytes)

        $$T_{msg} = t_s + t_b * L$$

        t_s and t_b can be obtained experimentally, by a ping-pong test and a linear regression
    - T_{free} : **free time,** when a PU becomes starved (without work)
      - Can be complex to measure since it depends on the order of tasks
      - Can be minimized with adequate load distribution and/or overlapping computation and communication

Performance of parallel applications

Optimisation: distributed memory (MPI) *vs.* shared memory (OpenMP)

- **Distributed memory (*vs.* shared memory)**
  - Data placement is explicit (*vs.* implicit)
  - Static scheduling is preferred (*vs.* dynamic)
  - Synchronization is costly (only performed by global barriers & message send)

- How to improve **scalability** on distributed memory?
  - Minimise communication among processes
    - Eventually duplicating computation
  - Minimise idle (free) time with a good load distribution

- **Practical advise**
  - Measure communication overhead
  - Measure load balance
  - Avoid centralised control