FACULTY
OF INFORMATICS
Masaryk University

PA039: Supercomputer Architecture and Intensive Computing

Message Passing Interface

Luděk Matyska

Spring 2024

Parallel programming

- Data parallelism
  - Identical instructions on different processors process different data
  - In principle the SIMD model (Single Instruction Multiple Data)
    - For example loop parallelization
- Task parallelism
  - MIMD model (Multiple Instruction Multiple Data)
  - Independent blocks (functions, procedures, programs) run in parallel
- SPMD
  - No synchronization at the level of individual instructions
  - Equivalent to MIMD
- Message passing targets SPMD/MIMD

Before MPI

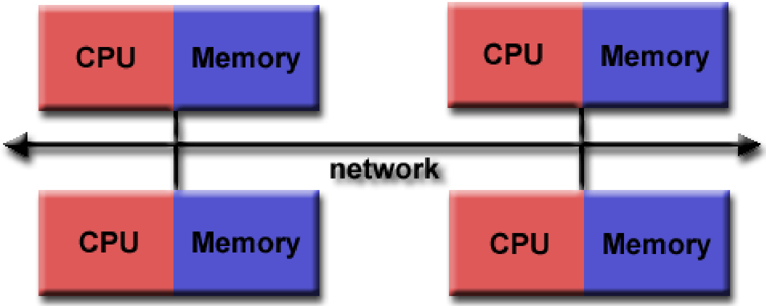
- Many competing message passing libraries
  - Vendor specific/proprietary libraries
  - Academic, narrow specific implementations
- Different communication models
  - Difficult application development
  - Need for “own” communication model to encapsulate the specific models
- MPI an attempt to define a standard set of communication calls

Message Passing Interface

- Communication interface for parallel programs
- Defined through API
  - Standardized
  - Several independent implementations
    - Potential for optimization for specific hardware
    - Some problems with real interoperability

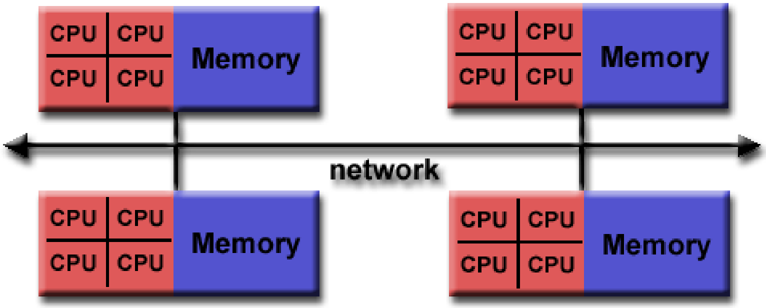
Programming model

MPI designed originally for distributed memory architectures



Programming model

Currently supports hybrid models



MPI Evolution

- Versions
  - 1.0 (1994)
    - Basic, never implemented
    - Bindings for C and Fortran
  - 1.1 (1995)
    - Removal of major deficiencies in Version 1.0
    - Implemented
  - 1.2 (1996)
    - Intermediate version (precedes MPI-2)
    - Extension of MPI-1 standard

MPI-2.0 (1997)

- Experimental Implementation of the full MPI-2 standard
- Extensions
  - Parallel I/O
  - Unidirectional operations (put, get)
  - Process manipulation
- Bindings for C++ and Fortran 90
- Stable for 10 years
  - Version 2.2 in 2009

MPI-3.0 (2012)

- Motivated by weaknesses of previous versions and also to reflect hardware innovation (esp. multicore processors), see `http://www.mpi-forum.org/`
- Major new features
  - Non-blocking collectives, neighbourhood collectives
  - Improved one-sided communication
  - New tools interface and bindings for Fortran 2008
- Other new features
  - Matching Probe and Recv for thread-safe probe and receive
  - New functions
  - Removed previously deprecated functions from C++ bindings
- Working groups
- MPI 3.1 ratified in June 2015
- Fully adopted in all major MPI implementations

MPI 4

- The current version
- Major additions:
  - “Big count” operations
  - Persistent Collectives
  - Partitioned Communication
  - Topology Solutions
  - Simple fault handling to enable fault tolerance solutions
  - New tool interface for events
- OpenMPI implementation
  - Currently Version 4.1 (approved in 2023)
  - Joint project of developers of several MPI streams

MPI Design Goals

- Portability
  - Define standard APIs
  - Define bindings for different languages
  - Independent implementations
- Performance
  - Independent hardware specific optimization
  - Libraries, potential for changes in algorithms
    - e.g. new versions of collective operations
- Functionality
  - Goal to cover all aspects of inter-processor communication

Design Goals II

- Library for message passing
- Designed for use on parallel computers, clusters and even Grids
- Make parallel hardware available for
    - Users
    - Libraries' authors
    - Tools and applications developers

Core MPI

MPI_Init	MPI Initialization
MPI_Comm_Size	Provide number of processes
MPI_Comm_Rank	Provide own (process) identity
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	MPI finish

MPI Initialization

- Create an environment
- Specify that the program will use the MPI libraries
- No explicit work with processes
    - Added since MPI-3.0

Identity

- Any parallel (distributed) program needs to know
  - How many processes are participating on the computation
  - Identity of “own” process
- MPI_Comm_size(MPI_COMM_WORLD, &size)
  - Returns number of processes that share the default MPI_COMM_WORLD communicator (see later)
- MPI_Comm_rank(MPI_COMM_WORLD, &rank)
  - Returns number of the calling process (identity)

Work with messages

- Naive/primitive model
  - Process A sends a message: operation *send*
  - Process B receives a message: operation *receive*
- Lot of questions
  - How to properly specify (define) the data?
  - How to specify (identify) process B (the receiver)?
  - How the receiver recognises that the data are for it?
  - How a successful completion is recognised?

Classical approach

- We send data as a byte stream
    - It is left to sender and receiver to properly setup and recognize data
- Each process has a unique identifier
    - We have to know identity of sender and receiver
    - Broadcast operation
- We can specify some tag for the better recognition (e.g. the message sequence number)
- Synchronization
    - Explicit collaboration between a sender and a receiver
    - It defines order of messages

Classical approach II

- send(buffer, len, destination, tag)
  - *buffer* contains data, its length is *len*
  - Message is sent to process whose identity is *destination*
  - Message has a tag *tag*
- recv(buffer, maxlen, source, tag, actlen)
  - Message will be accepted (read) into a memory space defined by the *buffer* whose length is *maxlen*
  - Actual size of accepted message is *actlen* ($actlen \leq maxlen$)
  - Message will arrive from a process with identifier *source* and must have a tag *tag*

Deficiencies of the classical approach

- Insufficient level of data specification/definition
  - Heterogeneity between sender and receiver (incompatible representation)
  - Too many copies
  - Too much relies on a programmer
- Tags are global
  - Complication when you want to write independent libraries
- Collective operations
  - too many send/receive operations
  - not optimized, inefficient

MPI extensions

- Processes are **grouped**
- Each message is defined within a specific **context** (not only a tag)
    - Messages could be sent and received only within the same context
- Group and context jointly define **communicator**
    - Tag is local to a specific communicator
- Default communicator **MPI_COMM_WORLD**
    - Group composed from all MPI processes
- Process identity (rank) is always defined within a specific context

Data types

- Data are described not by a tuple (address, length), but a triple (address, number, datatype)
- MPI Datatype is *recursively* defined as:
  - Pre-defined data type of the used language (e.g. MPI_INT)
  - Continuous array of MPI datatypes
  - Strided array of MPI datatypes
  - Indexed array of datatype blocks
  - Arbitrary datatype structure
- MPI provides functions to define own datatypes
  - e.g. a row of a matrix which is stored column-wise

Tags

- , Each message has an associated tag
  - Simplifies message recognition by the receiver
  - Tag is always defined within the used context (it is *scoped*)
- Receiver could specify which tag it expects
  - Alternatively it could ignore the tags (through MPI_ANY_TAG specification)

Point-to-point Communication

- Passing of a message between two processes
- Blocking / Non-blocking call (transmission)
  - Blocking – the call waits till the operation is finished
  - Non-blocking – the call just initiates the operation but does not wait till completion; the state of the data transfer must be tested independently
- Buffered / Un-buffered message passing
  - No buffer – message is passed directly without a buffer
  - MPI buffer – “transparent”, controlled directly by MPI
  - User buffer – controlled by the application (programmer)

Communication modes I

- Standard mode (Send)
  - Blocking call
  - MPI “decides”, if the MPI buffer is used
    - used → Send finishes when all data are in the buffer
    - not used → Send finishes when the data are accepted by the receiver
- Synchronous mode
  - Blocking call
  - Send finishes when the data were accepted by the receiver (processes synchronization)

Communication modes II

- Buffered mode
  - Buffer provided by the application(programmer)
  - Blocking or non-blocking – the operation finishes when the data are in the user buffer
- Ready mode
  - Receive must precede the actual send (Receive prepares the buffer)
  - Otherwise error

Basic *send* operation

- Blocking send
  - MPI_SEND(start, count, datatype, dest, tag, comm)
  - Triple (start, count, datatype) defines the message
  - *dest* identifies the receiver process, always relative to the used communicator *comm*
- Finishing the operation successfully means
  - All data were accepted by the system
  - The buffer is available for re-use
  - The receiver may not yet receive the data

Basic *receive* operation

- Blocking operation
  - MPI_RECV(start, count, datatype, source, tag, comm, status)
    - The operation waits till a message with a corresponding tuple (source, tag) is not received
    - *source* identifies the sending process, relative to the used communicator *comm*) or MPI_ANY_SOURCE
    - *status* contains info about the result of the operation
    - It also includes message tag and process identifier is MPI_ANY_TAG and MPI_ANY_SOURCE were used, resp.
    - If the accepted message contains less than *count* blocks, it is not interpreted as an error (the actual length is specified in the *status*)
    - Reception of more than *count* block is an error

Short Send/Receive protocol

- Fully duplex communication
  - Each sent message has a corresponding received message

-
  ```
  int MPI_Sendrecv(void *sendbuf, int sendcnt,
         MPI_Datatype sendtype, int dest, int sendtag,
         void *recbuf, int reccnt, MPI_Datatype recvtype,
         int source, int recvtag,
         MPI_Comm comm, MPI_Status *status)
  ```

Asynchronous communications

- Non-blocking *send* operation
  - Buffer can be re-used only after the completion of the whole transfer
- The *send* and *receive* operations create a request
  - Afterwards it is possible to check the status of the request
- Call

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

FACULTY
OF INFORMATICS
Masaryk University

Asynchronous operations II

- (Blocked) waiting for the operation to finish

  -
    ```
    int MPI_Wait(MPI_Request *request, MPI_Status *status)
    int MPI_Waitany(int cnt, MPI_Request *array_of_requests,
              int *index, MPI_Status *status)|
    int MPI_Waitall(int cnt, MPI_Request *array_of_requests,
              MPI_Status *array_of_statuses)
    ```

Asynchronous operation III

- Non-blocking status check

  -
    ```
    int MPI_Test(MPI_Request *request, int *flag,
              MPI_Status *status)
    int MPI_Testany(int cnt, MPI_Request *array_of_requests,
              int *flag, int *index, MPI_Status *status)
    int MPI_Testall(int cnt, MPI_Request *array_of_requests,
              int *flag, MPI_Status *array_of_statuses)
    ```
- Request release

  ```
  int MPI_Request_free(MPI_Request *request)
  ```

Persistent Communication Channels

- Non-blocking
- Created by combining two “half”-channels
- Life cycle
  - **Create (Start Complete)*** **Free**
  - Creation, followed by repetitive use, destroyed afterwards

Persistent channel – creation

```
int MPI_Send_init(void *buf, int cnt,
                  MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm,
                  MPI_Request *request)

int MPI_Recv_init(void *buf, int cnt,
                  MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm,
                  MPI_Request *request)
```

Transmission

- Transmission initialization (Start)

  -
    ```
    int MPI_Start(MPI_Request *request)
    int MPI_Startall(int cnt,
                     MPI_Request *array_of_requests)
    ```
- Finishing the transmission (Complete)
  - As in the asynchronous operations (wait, test, probe)

Channel destruction

- Equivalent to the destruction of the corresponding request

  `int MPI_Cancel(MPI_Request *request)`

Collective operations

- Operation performed by all processes within a group
  - Broadcast: `MPI_BCAST`
    - One process (root) will send data to all other processes
  - Reduction: `MPI_REDUCE`
    - Joins data from all processes in a group (communicator) and makes it available (as an array) to the calling process
  - Often a group of *send/receive* operations can be replaced by a single *bcast/reduce* operation
    - Higher efficiency/performance: *bcast/reduce* optimized for a particular hardware

Collective operations II

- Other operations
  - alltoall: exchange of messages among all processes in a group
    - *bcast/reduce* realizes the so called *scatter/gather* model
- Special reduction
  - min, max, sum, ...
  - User defined additional collective operations

Virtual topology

- MPI can define communication patterns that directly corresponds to the application needs
- These are (in a next step) mapped to the actual hardware ad its communication operations
  - Transparent
- Higher efficiency when writing programs
- Portability
  - Program is not directly associated with a concrete topology of used hardware
- Potential for independent optimizations

Date types

- Type Map
  - Typemap = $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$
- Type Signature
  - Typesig = $\{type_0, \ldots, type_{n-1}\}$
- Example:
  - MPI_INT == {(int,0)}

Extent and Size

- MPI_Type_extent(MPI_Datatype Type, MPI_Aint *extent)
- MPI_Type_size(MPI_Datatype Type, int *size)
- Example:
  - Type = {(double,0),(char,8)}
  - extent = 16
  - size = 9

Datatype construction

- Continuous data type

  -
    ```
    int MPI_Type_contiguous(int count,
                  MPI_Datatype oldtype,
                  MPI_Datatype *newtype)
    ```
- Vector

  -
    ```
    int MPI_Type_vector(int count, int blocklength,
                  int stride,
                  MPI_Datatype oldtype,
                  MPI_Datatype *newtype)
    int MPI_Type_hvector(int count, int blocklength,
                  int stride,
                  MPI_Datatype oldtype,
                  MPI_Datatype *newtype)
    ```

Datatype construction II

- Indexed data type

  -
    ```
    MPI_Type_indexed(int count, int *array_of_blocklengths,
           int *array_of_displacements, MPI_Datatype oldtype,
           MPI_Datatype *newtype)
    MPI_Type_hindexed(int count, int *array_of_blocklength,
           int *array_of_displacements, MPI_Datatype oldtype,
           MPI_Datatype *newtype)
    ```
- Structure

  -
    ```
    MPI_Type_struct(int count, int *array_of_blocklengths,
           MPI_Aint *array_of_displacements,
           MPI_Datatype *array_of_types,
           MPI_Datatype *newtype)
    ```

Datatype constructions III

- Confirmation of a datatype definition
  - `int MPI_Type_commit(MPI_Datatype *datatype)`
- Strided data types
  - They can include “holes”
  - Implementation may optimize some datatypes
  - Example: every second element of a vector
    - MPI could really “compose” a new data datatype
    - or it can send the whole vector and the selection is done at the receiver side

Operations over files

- Support since MPI-2
- File “parallelization”
- Basic terms
  - file
  - etype
  - view
  - file size
  - file handle
  - displacement
  - filetype
  - offset
  - file pointer

Operations over files II

Placement	Synch	Coordination	
		non-collective	collective
explicit offset	blocking	MPI_File_read_at	MPI_File_read_at_all
		MPI_File_write_at	MPI_File_write_at_all
	non-blocking & split collect.	MPI_File_iread_at	MPI_File_read_at_all_begin
			MPI_File_read_at_all_end
		MPI_File_iwrite_at	MPI_File_write_at_all_begin
			MPI_File_write_at_all_end
individual file ptrs	blocking	MPI_File_read	MPI_File_read_all
		MPI_File_write	MPI_File_write_all
	non-blocking & split collect.	MPI_File_iread	MPI_File_read_all_begin
			MPI_File_read_all_end
		MPI_File_iwrite	MPI_File_write_all_begin
			MPI_File_write_all_end
shared file ptr.	blocking	MPI_File_read_shared	MPI_File_read_ordered
		MPI_File_write_shared	MPI_File_write_ordered
	non-blocking & split collect.	MPI_File_iread_shared	MPI_File_read_ordered_begin
			MPI_File_read_ordered_end
		MPI_File_iwrite_shared	MPI_File_write_ordered_begin
	split collect.		MPI_File_write_ordered_end

MPI and optimizing compilers

- Asynchronous use of memory can lead to data changes (within arrays) that a complier knows nothing about
  - Copying of parameters will lead to loss of data

```
call user(a, rq)
call MPI_WAIT(rq, status, ierr)
write (*,*) a

subroutine user(buf, request)
call MPI_IRECV(buf,...,request,...)
end
```

  - In this example, main program will print a non-sensical value of “a” as the return from “user” the actual value of “a” will be copied while the corresponding *receive* operation may not be finished yet