HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

High Performance Computing

Lecture 1 - Introduction to HPC and MPI

Felicja Okulicka-Dłużewska

F.Okulicka@mini.pw.edu.pl

Warsaw University of Technology

Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY
DEVELOPMENT PROGRAMME

Go to full-screen mode now by hitting CTRL-L

Overview

- **Introduction to HPC**
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

HPC

HPC = high performance computing
≡ large scale computation ≡ parallel computing

- To be run using multiple CPUs.
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

Examples

- Galaxy formation
- Planetary movement
- Weather and ocean patterns
- Tectonic plate drift
- Rush hour traffic
- Automobile assembly line
- Building a space shuttle
- Ordering a hamburger at the drive through.
- Databases, data mining

Examples

- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Management of national and multi-national corporations
- Financial and economic modeling
- Advanced graphics and virtual reality, particularly in the entertainment industry

Why Use Parallel Computing ?

Parallelism is the future of computing.

Flynn's Taxonomy

- **Single Instruction, Single Data** (SISD): A serial (non-parallel) computer
- **Single Instruction, Multiple Data** (SIMD): A type of parallel computer
- **Multiple Instruction, Single Data** (MISD): theoretical only
- **Multiple Instruction, Multiple Data** (MIMD): the most common type of parallel computer, for example clusters
  - Shared memory systems
  - Distributed memory systems

Parallel Terminology

- **Task**: A logically discrete section of computational work: typically a program or program-like set of instructions that is executed by a processor.
- **Parallel Task**: A task that can be executed by multiple processors safely (yields correct results)

Program execution

- **Serial Execution**: Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.
- **Parallel Execution**: Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
- **Pipelining**: Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Parallel Computers

- **Shared Memory**: a computer architecture where all processors have direct (usually bus based) access to common physical memory
- **Symmetric Multi-Processor (SMP)**: multiple processors share a single address space and access to all resources; shared memory computing.
- **Distributed Memory**: In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Parallel Terminology

- **Communications**: Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.
- **Synchronization**: The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Parallel Terminology

- **Granularity**: is a qualitative measure of the ratio of computation to communication.
  - Coarse: relatively large amounts of computational work are done between communication events
  - Fine: relatively small amounts of computational work are done between communication events
- **Observed Speedup**:the simplest and most widely used indicators for a parallel program's performance:

$$\frac{wall-clock-time-of-serial-execution}{wall-clock-time-of-parallel-execution}$$

Parallel Terminology

- **Scalability**: Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include: Hardware - particularly memory-cpu bandwidths and network communications
- **Application algorithm** Characteristics of your specific application and coding

Parallel machines

- **Multi-core Processors**: Multiple processors (cores) on a single chip.
- **Cluster Computing**
- **Supercomputing** / **High Performance Computing** : Use of the world's fastest, largest machines to solve large problems.

Memory

Parallel Computer Memory Architectures:

- Shared Memory
- Uniform Memory Access (UMA): Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Non-Uniform Memory Access (NUMA): Often made by physically linking two or more SMPs

Models

Parallel programming models in common use:

- Shared Memory
- Threads :
  - POSIX Threads
  - OpenMP
- Message Passing
- Data Parallel
  - High Performance Fortran (HPF)
  - Compiler Directives
- Hybrid

Designing Programs

Designing Parallel Programs:

- Automatic
- Manual Parallelization
  - Loop parallelization (splitting)
  - Partitioning the problem:
    - domain decomposition
    - functional decomposition.

Communications

- Embarrassingly parallel problems - no communication - they are so straight-forward. Very little inter-task communication is required.
- Others - the cost of communication is important

Communication

Cost of communication:

- Inter-task communication virtually always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Communications

Latency vs. Bandwidth:

Latency is the time it takes to send a minimal (0 byte) message from point A to point B.
Commonly expressed as microseconds.

Bandwidth is the amount of data that can be communicated per unit of time.
Commonly expressed as megabytes/sec or gigabytes/sec.

Communications

Synchronous vs. asynchronous communications :

- Synchronous communications require some type of "handshaking" between tasks that are sharing data.
- Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from one another.

Communications

Synchronous vs. asynchronous communications :

- Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place.
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Communications

Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.
Both of the **two scopings** can be implemented synchronously or asynchronously:

- Point-to-point
- Collective

Synchronization

Synchronization:

- Barrier
- Lock / semaphore
- Synchronous communication operations

Designing Programs

Designing Parallel Programs the following factors should be taken under considaration:

- data dependencies
- Load Balancing
- Granularity: Computation / Communication Ratio
- I/O

Spped-up

Amdahl's Law states that potential program speedup is defined by the fraction of code (p) that can be parallelized.

Each algorithm contains parts which can not be parallelized.

Let:

s - sequential part of the algorithm

p - parallel part of the algorithm.

We have:

$$s + p = 1$$

Amdahl's theorem

We can express the system time on one processor by

$$t_1 = s + p = 1$$

and the system time on P processors by

$$t_P = s + \frac{p}{P}$$

. An upper bound for the maximally available speedup is following:

$$S_{P,max} = \frac{t_1}{t_P} = \frac{1}{(s + \frac{(1-s)}{P})} \leq \frac{1}{s}$$

because $\frac{(1-s)}{P} \geq 0$

Complexity

The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

Parallel software

The most popular libraries:

- POSIX Threads
- OpenMP
- Parallel Virtual Machine (PVM)
- Message Passing Interface (MPI)

Overview

- Introduction to HPC
- **Introduction to MPI**
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

MPI

Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers.

Goals and Scope of MPI are:

- To provide source-code portability
- To allow efficient implementation
- To offer a great deal of functionality
- To offer support for heterogeneous parallel architecture

Model of MPI

Visibility of communications:

- With the Message Passing Model communications are explicit and generally quite visible and under the control of the programmer.
- With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

MPI

Model of computation in MPI:

- Portability
- Resource Requirements
- Scalability: The ability of a parallel program's performance to scale is a result of a number of interrelated factors.

MPI Model

By MPI the SPMD (Single Program, Multiple Data) with the message passing is simulated.

Each processor in a message passing program runs a process - the sub-program written in a conventional sequential language.

All variables are private - there is no common memory.

Compilation

The compilation:

mpicc program-name

mpicf90 program-name

mpicf77 program-name

Running

To start the MPI program, normally mpirun(MPI-1) or mpiexec (MPI-2) is called:

```
mpiexec -n <numprocs> <program>
mpirun -n <numprocs> <program>
```

The number of processes <numprocs> are generated:

Other arguments to mpiexec may be implementation-dependent.

MPI headers

Header files must be included:

in C:

include $< mpi.h >$

in FORTAN:

include 'mpif.h'

Overview

- Introduction to HPC
- Introduction to MPI
  - **Functions**
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

MPI functions

Functions are called as follows:

in C:

```
Error=MPI_xxxx(parameter,...);
```

in FORTAN:

```
CALL MPI_xxxx(parameter,..., IError)
```

Types of MPI Calls

When discussing MPI procedures the following terms are used:

- local
- non-local
- blocking
- nonblocking
- collective

Types of MPI Calls

- local

If the completion of the procedure depends only on the local executing process. Such an operation does not require an explicit communication with another user process. MPI calls that generate local objects or query the status of local objects are local.

- non-local

If completion of the procedure may require the execution of some MPI procedure on another process. Many MPI communication calls are non-local.

Types of MPI Calls

- blocking

If return from the procedure indicates the user is allowed to re-use resources specified in the call. Any visible change in the state of the calling process affected by a blocking call occurs before the call returns.

Types of MPI Calls

- nonblocking

If the procedure may return before the operation initiated by the call completes, and before the user is allowed to re-use resources (such as buffers) specified in the call. A nonblocking call may initiate changes in the state of the calling process that actually take place after the call returned: e.g. a nonblocking call can initiate a receive operation, but the message is actually received after the call returned.

- collective

If all processes in a process group need to invoke the procedure.

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - **MPI program**
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

MPI programs

MPI controls its own internal data structures and releases 'handlers' to allow programmers to refer to these.

C handles are of defined typedefs whereas Fortran handles are INTEGERs.

MPI_Init

The routine for initializing MPI must be called as the first MPI function:

in C:

```
Int MPI_Init(int *argc, char ***argv)
```

in FORTAN:

```
MPI_INIT(IError)
```

It creates the MPI_COMM_WORLD communicator as the environment for communication between the group of processes.

MPI_COMM_RANK

To identify different processes the rank is used. The rank is returned by the routine:

In C:

MPI_Comm_rank(MPI_Comm comm, int *rank)

in FORTAN:

MPI_COMM_RANK(integer comm, integer irank, integer ierror)

Communicator's Size

By calling

in C:

MPI_Comm_size(MPI_Comm comm, int *size)

In FORTAN:

MPI_COMM_SIZE(integer comm, integer isize, integer ierror)

the number of processes contained within a communicator is obtained.

Exiting MPI

Exiting MPI routine must be called by all processes:

In C:

```
int MPI_Finalized(int *flag)
```

In Fortran:

```
MPI_FINALIZED(logical FLAG, integer ierror)
```

In C++:

```
bool MPI::Is_finalized()
```

Example 1

```
#include <stdio.h>
#include <mpi.h>
int main( int argc, char argv[] )
{
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) printf("Starting program ");
    MPI_Finalize();
    return 0;
}
```

Example 2

```
/* "Hello, parallel worlds!" */
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[] )
{
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf( "Hello, parallel worlds! It's me: processor %d\n ",
                                myrank );
    MPI_Finalize();
    return 0;
}
```

Example 2 - results

You can expect to see something like thiswhen you execute the program on 4 processors:

Hello, parallel worlds! It's me: processor 1!
Hello, parallel worlds! It's me: processor 3!
Hello, parallel worlds! It's me: processor 4!
Hello, parallel worlds! It's me: processor 2!

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - **Messages**
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

Messsages in MPI

Messages are packets of data moving between subprograms.

The message passing system has the following information:

1. sending processor
2. source location
3. data type
4. data length
5. receiving processor(s)
6. destination location
7. destination size

The receiving process is capable of dealing with messages which are sent.

Example

```
MPI_Send( msg,
          SIZE,
          MPI_DOUBLE,
          dest,
          tag,
          MPI_COMM_WORLD );
```

Communication

There are several **types of message passing**

- point-to point communication
- collective communication
- Broadcasting

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - **Point-to-point communication**
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - Data types

Point-to point comm.

Point-to-point communication:

- simplest form of message passing
- one process sends a message to another
- there are different types of point-to-point communication

Point-to point comm.

Point-to-point comm.:

- **synchronous send** - provide information about the completion of the message
- **asynchronous send** - only know when the message has left
- **blocking operations**: relate to when the operation has completed only return from the subroutine call when the operation has completed
- **non-blocking operations**

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - **Non-blocking operations**
  - Collective communication
  - Standard Send/Receive
  - Data types

Non-blocking oper.

- Return strait away and allow the sub-program to continue to perform other work. At some time later the subprogram can test or wait for the completion of the non-blocking operation
- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - **Collective communication**
  - Standard Send/Receive
  - Data types

Collective comm.

Collective communication routines are higher level routines involving several processes at a time.
Can be built out of point-to-point communications

Collective communications:

- **Barriers** - synchronize processes.
- **Broadcast** - a one-to-many communication
- **Reduce operations** - Combine data from several processes to produce a single result

Modes

There are different type of **communication modes**:

- **Synchronous send** - only completes when the receive has completed: **MPI_SSEND**
- **Buffered send** - always completes (unless an error occurs), irrespective of receiver **MPI_BSEND**
- **Standard send** - aither synchronous or buffered **MPI_SEND**
- **Ready send** - always completes (unless an error occurs), irrespective or whether the receive has completed **MPI_RSEND**
- **Receive** - completes when a message has arrived **MPI_RECV**

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - **Standard Send/Receive**
  - Data types

Standard MPI_Send

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

MPI_Recv

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

Rules

A sub-program needs to be connected to a message passing system.

Messages need to have addresses to be sent to.

The receiving process is capable of dealing with messages it is sent.

MPI_SEND / MPI_RECV form the blocked communication - the processes wait until the communication is completed.

Send in Fortran

Standard send in FORTRAN:

```
MPI_SEND(
        <type> buf,
        integer count,
        MPI_Datatype datatype,
        integer dest,
        integer tag,
        integer comm,
        integer ierror)
```

Receive in Fortran

```
MPI_RECV(
        <type> buf,
        integer count,
        MPI_Datatype datatype,
        integer source,
        integer tag,
        integer comm,
        integer status,
        integer ierror)
```

MPI_Buffer_attach

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

In C:

int MPI_Buffer_attach(void* buffer, int size)

In FORTRAN:

MPI_BUFFER_ATTACH(<type>buffer,integer size, integer ierror)

Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages.

The buffer is used only by messages sent in buffered mode.

Only one buffer can be attached to a process at a time.

MPI_Buffer_dettach

In C:

int MPI_Buffer_detach(void* buffer_addr, int* size)

In FORTRAN:

MPI_BUFFER_DETACH(<type>buffer_addr,integer size, integer ierror)

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer.

Example 3

```
/* Example Calls to attach and detach buffers */
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

Status in MPI_RECV

Communication envelope is returned from MPI_RECV as status. Information includes:

- source status.MPI_SOURCE
- tag status.MPI_TAG
- count MPI_GET_count

int MPI_Get_status(MPI_Status status, MPI_Datatype datatype, int *count)
int MPI_Request_get_status(MPI_Request request, int *flag, MPI_Status *status)
bool MPI::Request::Get_status(MPI::Status& status) const
bool MPI::Request::Get_status() const

Message order

Message order preservation

- messages do not overtake each other
- this is true even for non-synchronous sends

Success in comm.

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message types must match
- Receiver buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source: MPI_ANY_SOURCE
- To receive with any tag: MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter

Overview

- Introduction to HPC
- Introduction to MPI
  - Functions
  - MPI program
  - Messages
  - Point-to-point communication
  - Non-blocking operations
  - Collective communication
  - Standard Send/Receive
  - **Data types**

Datatypes in MPI

A message contains a number of elements of some particular datatypes. MPI datatypes:

1. Basic types
2. derived types
   - vectors
   - structs
   - others

Derived types can be built up from basic types

C types are different from Fortran types

Datatypes in MPI

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int

Datatypes in MPI

MPI datatype	C datatype
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

Thank you for your attention!

Any questions are welcome.