Introduction to MPI
Parallel Programming with the Message Passing Interface

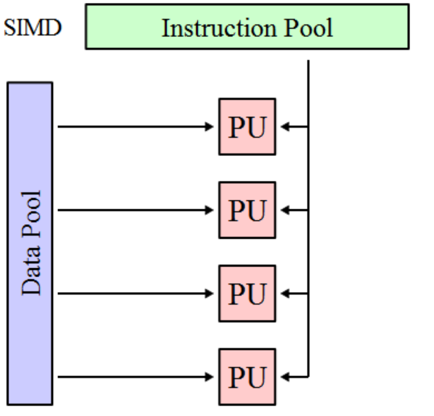
Professor: Panagiota Fatourou
TA: Eleftherios Kosmas
CSD - March 2012

Message Passing Model

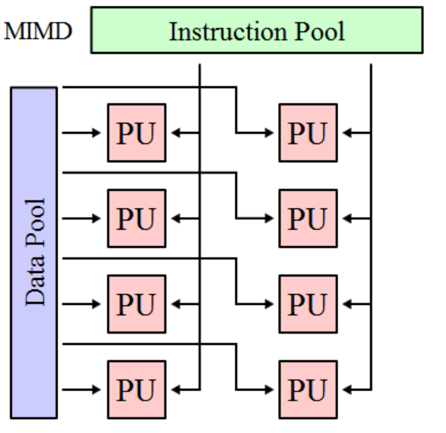
- process: a program counter and an address space
  - may have multiple threads (program counters and associated stacks) sharing a single address space
- MPI: communication among processes, which have separate address spaces
- Interprocess communication consists of
  - **Synchronization**
  - **Movement of data** from one process's address space to another's

Types of Parallel Computer Models

- **Data Parallel** - the same instructions are carried out simultaneously on multiple data items (**SIMD**)



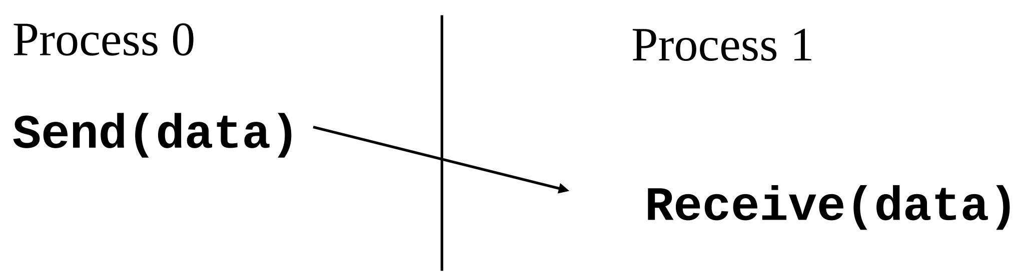
- **Task Parallel** - different instructions on different data (**MIMD**)
  - **SPMD** (single program/process, multiple data)
    - execute the same program at independent points,
    - not in the lockstep that SIMD imposes on different data



- Message Passing (and MPI) is for MIMD/SPDM parallelism

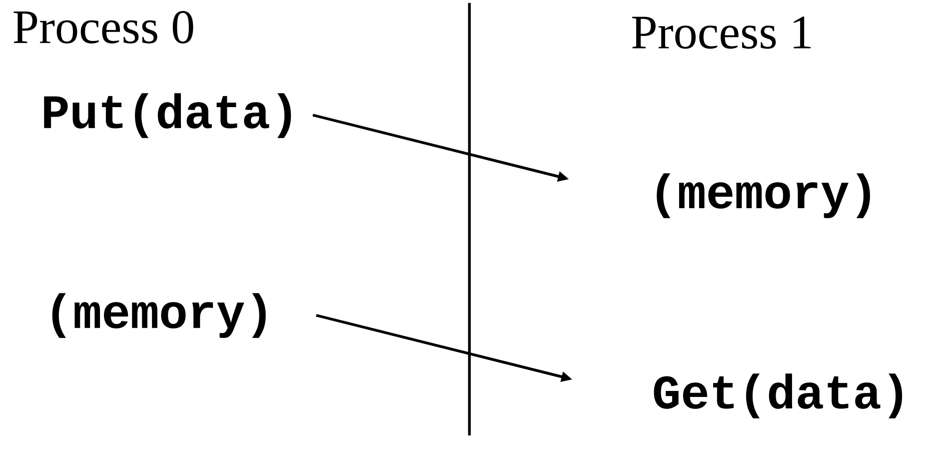
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data **cooperative**
  - Data is explicitly sent by one process and received by another
  - any change in the receiving process's memory is made with the receiver's explicit participation
- communication and synchronization are combined



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
  - Only one process needs to explicitly participate
  - communication and synchronization are decoupled
- One-sided operations are part of MPI-2



What is MPI?

- A message-passing library specification
  - ✓ extended message-passing model
  - ✗ not a language or compiler specification
  - ✗ not a specific implementation or product

- For parallel computers, clusters, and heterogeneous networks

- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Why use MPI?

- MPI provides a **powerful**, **efficient**, and **portable** way to express parallel programs
- MPI was explicitly designed to enable **libraries**…
- … which may **eliminate the need** for many users to learn (much of) MPI

How to use MPI?

- The MPI-1 Standard does not specify how to run an MPI program
  - it is dependent on the implementation of MPI you are using
  - might require various scripts, program arguments, and/or environment variables

- So, MPI-1 does not provide mechanisms to manipulate processes
  - ✍ Note: related functions have been added to MPI-2, e.g., `MPI_Comm_Spawn()`

- most implementations use some external application, e.g., `mpirun`
  - in order to create 10 processes that execute the same program `myprog`, we execute:

```
mpirun -np 10 myprog
```

A Minimal MPI Program (C)

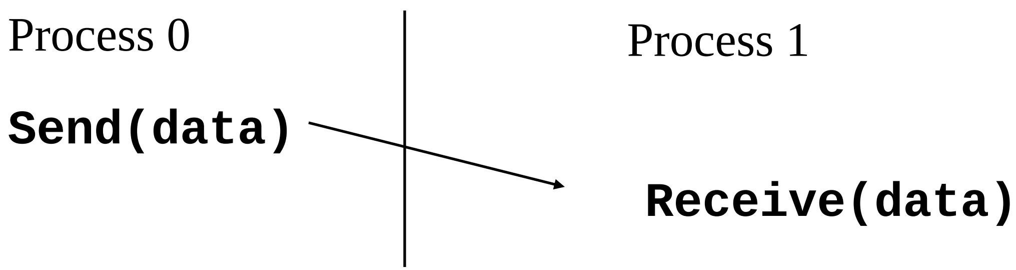
```
#include <mpi.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

- All process must use
  - **MPI_Init**, to initialize the MPI execution environment
  - **MPI_Finalize**, to finalize it

MPI Basic Send/Receive

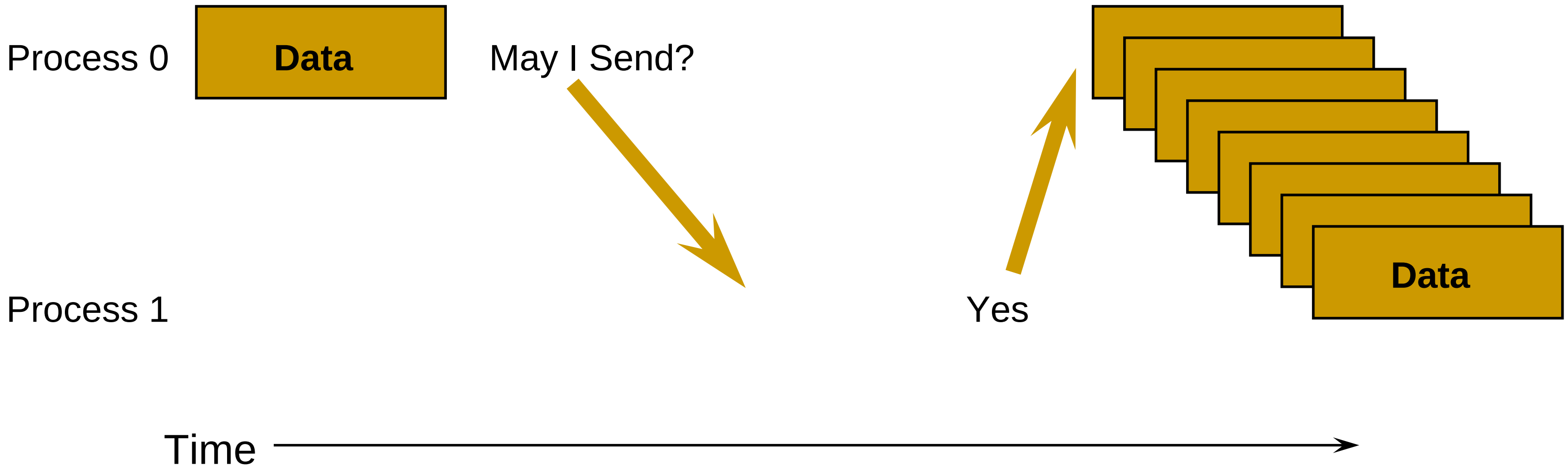
- We need to fill in the details in



- Things that need specifying:
  - How will "data" be **described**?
  - How will processes be **identified**?
  - How will the receiver **recognize/screen messages**?
  - What will it mean for these operations to **complete**?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Some Basic Concepts

- Processes can be collected into **groups**
- Each message is sent in a **context**, and must be received in the same context
- A group and context together form a **communicator**
  - There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**
- A process is identified by its **rank** in the group associated with a communicator

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- ☞ MPI provides functions to answer these questions
- If we create n processes, then in each of them will be assigned a **unique** identifier from **`0,1,…,n-1`**
- **`MPI_Comm_size (comm, &size)`**
  - reports the number of processes
  - `comm`, `MPI_COMM`
- **`MPI_Comm_rank (comm, &rank)`**
  - reports the `rank`, a number between 0 and `size`-1, identifying the calling process

Better Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );

    MPI_Finalize();
    return 0;
}
```

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., `MPI_INT`, `MPI_DOUBLE_PRECISION`)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, e.g.,
  - an array of (`int`, `float`) pairs, or
  - a row of a matrix stored columnwise

MPI Tags

- Messages are sent with an accompanying **user-defined** integer tag
  - assist the receiving process in identifying the message
- Messages
  - can be screened at the receiving end by specifying a specific tag,
  - or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive

✍ Note:
  - Some non-MPI message-passing systems have called tags “message types”
  - MPI calls them tags to avoid confusion with datatypes.

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`, which is the rank of the target process in the communicator specified by `comm`
- **blocks** until
  - the data has been delivered to the system and the buffer can be reused
- when it returns,
  - the message may not have been received by the target process
- `datatype`, `MPI_Datatype`

MPI Basic (Blocking) Receive

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

- Blocks until
  - a matching (on `source` and tag) message is received from the system, and
  - the buffer can be used
- `source` is rank in communicator specified by `comm`, or `MPI_ANY_SOURCE`.
- `status` contains further information
- Receiving fewer than count occurrences of datatype is OK,
  - but receiving more is an error

Retrieving Further Information

- **status** is a data structure allocated in the user's program

```
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 );
```

Example

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

int main(int argc, char *argv[]) {
   int numtasks, rank, dest, source, rc, count, tag=1;
   char inmsg, outmsg='x';
   MPI_Status Stat;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0) {
      dest = 1;
      source = 1;
      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
           &Stat);
   } ...
```

Example

```
...
   else if (rank == 1) {
        dest = 0;
        source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
               MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
               MPI_COMM_WORLD);
   }

   rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
   printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

   MPI_Finalize();
}
```

Why Datatypes?

- all data are labeled by type
- an MPI implementation can support heterogeneous communication, between processes on machines with
  - different memory representations
    - e.g., Little-Endian, Big-Endian
  - lengths of elementary datatypes
- Specifying application-oriented layout of data in memory
  - **reduces** memory-to-memory **copies** in the implementation
  - allows the use of special hardware (**scatter/gather**) when available
    - the process of gathering data from, or scattering data into, a given set of buffers

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "**wild card**" tags
- Contexts are different from tags
  - no wild cards allowed
  - **allocated dynamically** by the system when a library sets up a communicator for its own use
- User-defined tags still provided in MPI for user convenience in organizing application
- `MPI_Comm_split` creates new communicators

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_SEND**
  - **MPI_RECV**
- Point-to-point (send/recv) isn't the only way...

Collective Operations in MPI

- Collective operations are called by all processes in a communicator
- `MPI_BCAST (&buf, count, datatype, root, tag, comm)`
  - distributes data from one process (`root`) to all others in the communicator `comm`
- `MPI_REDUCE (&sendBuf, &recvBuf, count, datatype, op, root, comm)`
  - combines data from all processes in communicator (`comm`) and returns it to one process (`root`)
  - applies operation `op` (`MPI_Op`)
    - e.g., `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, `MPI_MAXLOC`

- In many numerical algorithms, `SEND/RECEIVE` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency.

Example - PI

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] ) {
  int n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (1) {
     if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     ...
```

Example - PI

```
    ...
    if (n == 0) break;
    else {
       h   = 1.0 / (double) n;
       sum = 0.0;
       for (i = myid + 1; i <= n; i += numprocs) {
          x = h * ((double)i - 0.5);
          sum += (4.0 / (1.0 + x*x));
       }
       mypi = h * sum;
       MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                  MPI_COMM_WORLD);
       if (myid == 0)
          printf("pi is approximately %.16f, Error is %.16f\n",
                      pi, fabs(pi - PI25DT));
       }
   }
   MPI_Finalize();
   return 0;
}
```

Collective Operations in MPI

- **MPI_Allreduce (&sendBuf, &recvBuf, count, datatype, op, comm)**
  - combines data from all processes in communicator (comm) and send the to all processes
- **MPI_Op_create (&function, commute, &op)**
  - creates a user-defined combination function handle
  - **function**, MPI_User_Function
  - **commute**, integer, equals 1 if operation is commutative

```
typedef void (MPI_User_function) ( void *a,
                     void *b, int *len, MPI_Datatype * );
```

  - where the operation is: **b[i] = a[i] op b[i], for i=0,...,len-1**
- **MPI_Op_free (&op)**
  - frees a user-defined combination function handle

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This is called "unsafe" because it depends on the availability of system buffers

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(1)	**Irecv(0)**
Waitall	**Waitall**

MPI Non-Blocking Send/Recv

- `MPI_Isend (&buf, count, datatype, dest, tag, comm, &request)`
- `MPI_Irecv (&buf, count, datatype, source, tag, comm, &request)`
  - change: status of `MPI_Recv` is missing
  - ☞ `request`, `MPI_Request`, is used in order to examine if operation is completed

- `MPI_Test(&request, &flag, &status)`
  - non-blocking
  - `flag`, integer, equals 1 if operation is completed
- `MPI_Wait(&request, &status)`
  - blocking

MPI Non-Blocking Send/Recv

- **`MPI_Testany (count, &array_of_requests, &index, &flag, &status)`**
  - ☞ tests for completion of any previously initiated communication
  - ☞ non-blocking
  - ❑ `count`, list length
  - ❑ `index`, index of operation completed, or `MPI_UNDEFINED` if none completed
  - ❑ `flag`, equals 1 if one of the operation is completed

- **`MPI_Waitany (count, &array_of_requests, &index, &status)`**
  - ☞ blocking

Useful MPI Functions

- **`MPI_Barrier (comm)`**
  - blocks the caller until all group members have called it
  - it returns at any process only after all group members have entered the call
- **`MPI_Get_processor_name (&name, &len)`**
  - a process can get the name of the processor
  - **`name`**, must be an array of size at least `MPI_MAX_PROCESSOR_NAME`
  - **`len`**, length of the name
- **`MPI_Wtime()`**
  - Time in seconds since an arbitrary time in the past
- **`MPI_Wtick()`**
  - returns the resolution of **`MPI_Wtime()`** in seconds

Error Handling

- By default, an error causes all processes to abort
  - a default error handler is called that aborts the MPI job
- The user can cause routines to return (with an error code) instead
- A user can also write and install custom error handlers
  - `MPI_Errhandler_set`
- The reduction function (e.g., `MPI_MAX`) do not return an error value. Upon an error,
  - either call `MPI_Abort`, or
  - silently skip the problem

- Libraries might want to handle errors differently from applications

Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
    - e.g., `MPI_Comm_spawn()`, `MPI_Comm_get_parent()`
  - Dynamic establishment of connections
    - e.g., `MPI_Comm_connect()`, `MPI_Comm_accept()`, `MPI_Open_port()`, `MPI_Close_port()`, `MPI_Publish_name()`, `MPI_Unpublish_name()`, `MPI_Lookup_name()`
    - Similar to TCP/IP sockets / DNS lookups
    - `MPI_Comm_join()`
    - `MPi_Comm_disconnect()`
- One-sided communication
  - `MPI_Put()` / `MPI_Get()`
  - `MPI_Win_create()`, `MPI_Win_fence()`, `Mpi_Win_free()`
- Parallel I/O
  - e.g., `MPI_File_open()`, `MPI_File_read_at()`, `MPI_File_write_at()`, `MPI_File_set_atomicity()`

Compiling and Executing

- Create a file `hosts` including the names of host machines

```
milo:~/CS556/mpi> mpicc pi.c -o pi
milo:~/CS556/mpi> cat hosts
milo
fraoula
milo:~/CS556/mpi> mpirun -np 10 -hostfile hosts pi
Enter the number of intervals: (0 quits) 100000
pi is approximately 3.1415926535981269, Error is
  0.0000000000083338
Enter the number of intervals: (0 quits) 0
```

Configuring

- First time setup your ssh public keys, to login to the nodes without password

  i. `ssh-keygen -t dsa`

  ii. into `.ssh` directory make: `cat id_dsa.pub > authorized_keys`

- Also it is recommended to login once to each node that you will use with mpi

More information?

- http://www.mcs.anl.gov/research/projects/mpi/

The End - Questions