Programming Scalable Systems with MPI

Clemens Grelck

University of Amsterdam

UvA / SURFsara
High Performance Computing and Big Data

Programming Scalable Systems with MPI

Targeted Systems: Clusters and Supercomputers

Characteristics:

- Many (usually) identical machines (*compute nodes*)
- High-speed network (e.g. Infiniband)
- Loosely coupled
- Distributed memory architecture

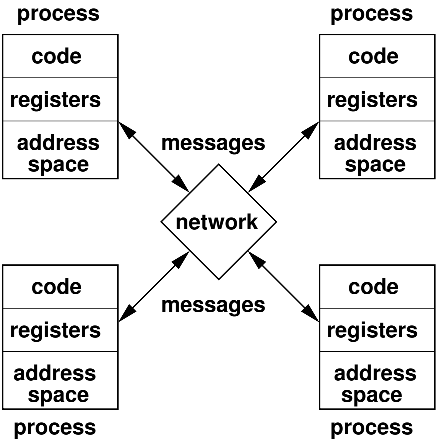
Examples:

Tianhe-2
NUDT, China
TOP500 #1

Sequoia
LLNL, USA
TOP500 #3

Message Passing as a Programming Paradigm

Programming model:



Distributed memory architectures !

Message Passing as a Programming Paradigm

Core idea:

- Code for individual processes written in sequential language
- Ability to send and receive messages provided via library
- Know who you are and who else is out there

Message Passing as a Programming Paradigm

Core idea:

- Code for individual processes written in sequential language
- Ability to send and receive messages provided via library
- Know who you are and who else is out there

Applicability:

- Designed for network-connected sets of machines
- Applicable to shared memory architectures as well
- Applicable to uniprocessor with multitasking operating system

Message Passing as a Programming Paradigm

Core idea:

- Code for individual processes written in sequential language
- Ability to send and receive messages provided via library
- Know who you are and who else is out there

Applicability:

- Designed for network-connected sets of machines
- Applicable to shared memory architectures as well
- Applicable to uniprocessor with multitasking operating system

Characterisation:

- Very low-level and machine-oriented
- Deadlocks: wait for message that never comes
- Unstructured (*spaghetti*) communication:
  Send/receive considered the *goto of parallel programming*

Programming Scalable Systems with MPI

What is MPI ?

MPI is NOT a library !

What is MPI ?

MPI is NOT a library !

MPI is a specification !

- Names of data types
- Names of procedures (MPI-1: 128, MPI-2: 287)
- Parameters of procedures
- Behaviour of procedures

What is MPI ?

MPI is NOT a library !

MPI is a specification !

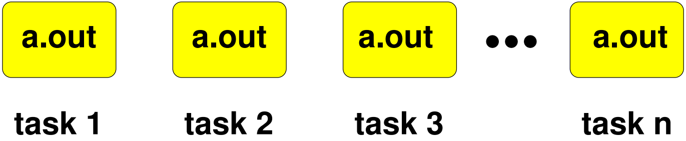
- Names of data types
- Names of procedures (MPI-1: 128, MPI-2: 287)
- Parameters of procedures
- Behaviour of procedures

Bindings for different languages:

- Fortran
- C
- C++ (MPI-2 only)

Organization Principle of MPI Programs

SPMD — Single Program, Multiple Data:



Characteristics:

- Each task executes the same binary program.
- Tasks may identify total number of tasks.
- Tasks may identify themselves.
- All tasks are (implicitly) created at program startup.
- Specific program launcher: `mpirun`
- All tasks are (implicitly) shut down at program termination.

My First MPI Program: Distributed Hello World

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

int main( int argc, char *argv[])
{
  int rc, num_tasks, my_rank;

  rc = MPI_Init( &argc, &argv);                    // Init runtime

  if (rc != MPI_SUCCESS) {                         // Success check
    fprintf( stderr, "Unable to set up MPI");
    MPI_Abort( MPI_COMM_WORLD, rc);                // Abort runtime
  }

  MPI_Comm_size( MPI_COMM_WORLD, &num_tasks); // Get num tasks
  MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);   // Get task id

  printf( "Hello World says %s!\n", argv[0]);
  printf( "I'm task number %d of a total of %d tasks.\n",
          my_rank, num_tasks);

  MPI_Finalize();                                  // Shutdown runtime
  return 0;
}
```

Compiling First MPI Program

HowTo:

```
mpicc  -o hello_world hello_world.c  // for C
mpicxx -o hello_world hello_world.c  // for C++ programs
mpif77 -o hello_world hello_world.c  // for Fortran77 programs
mpif90 -o hello_world hello_world.c  // for Fortran90/95 programs
```

mpiXYZ are compiler wrappers:

- set paths properly
- link with correct libraries
- ...

Running First MPI Program

Example output:

```
grelck@das4:> mpirun -n 8 hello_world
  Hello World says hello_world!
  Hello World says hello_world!
  Hello World says hello_world!
  Hello World says hello_world!
  I'm task number 4 of a total of 8 tasks.
  Hello World says hello_world!
  I'm task number 5 of a total of 8 tasks.
  I'm task number 6 of a total of 8 tasks.
  Hello World says hello_world!
  I'm task number 7 of a total of 8 tasks.
  Hello World says hello_world!
  I'm task number 0 of a total of 8 tasks.
  I'm task number 3 of a total of 8 tasks.
  Hello World says hello_world!
  I'm task number 1 of a total of 8 tasks.
  I'm task number 2 of a total of 8 tasks.
```

Essential MPI Routines: MPI_Init

Signature:

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

Characteristics:

- Initializes MPI runtime system.
- Must be called by each process.
- Must be called before any other MPI routine.
- Must be called exactly once.
- Distributes command line information.
- Returns error condition.

Essential MPI Routines: MPI_Finalize

Signature:

```
int MPI_Finalize( void)
```

Characteristics:

- Finalizes MPI runtime system.
- Must be called by each process.
- Must be called after any other MPI routine.
- Must be called exactly once.
- Returns error condition.

Essential MPI Routines: MPI_Abort

Signature:

```
int MPI_Abort( MPI_Comm communicator, int error_code)
```

Characteristics:

- Aborts program execution.
- Shuts down ALL MPI processes.
- More precisely:
  shuts down all processes referred to by `communicator`.
- Replaces `MPI_Finalize`.
- Must be used instead of `exit` or `abort`.
- MPI process system returns `error_code` to surrounding context.
- Standard communicator: `MPI_COMM_WORLD`

Essential MPI Routines: MPI_Comm_size

Signature:

```
int MPI_Comm_size( MPI_Comm comm,   \\ IN  : communicator
                   int *size        \\ OUT : number of tasks
)
```

Characteristics:

- Queries for number of MPI processes.
- More precisely: size of "`communicator`".
- Result is "returned" in parameter "`size`".
- Returns error condition.

Essential MPI Routines: MPI_Comm_rank

Signature:

```
int MPI_Comm_rank( MPI_Comm comm,     \\ IN  : communicator
                   int *rank          \\ OUT : task id
)
```

Characteristics:

- Queries for task ID, called "rank".
- More precisely: task ID with respect to "communicator".
- Result is "returned" in parameter "rank".
- Returns error condition.

MPI Routines

Common design characteristics:

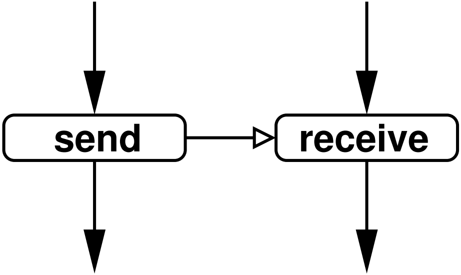
- All routine names start with "MPI_".
- Name components are separated with underscores.
- First component starts with upper case letter.
- All routines return integer error code.
  - `MPI_SUCCESS`
  - `MPI_ERR_XXX`
- Routines have 3 types of parameters:
  - IN: regular parameter, read by routine.
  - OUT: return parameter, written by routine.
  - INOUT: reference parameter, read and written by routine.

Programming Scalable Systems with MPI

Scope of Communication

Point-to-Point Communication:

- ONE Sender
- ONE Receiver
- ONE Message



Introductory Example

Algorithmic idea:

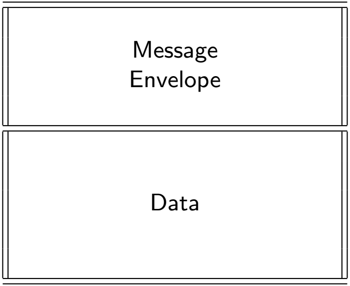
- Task #0 sends some string to task #1.
- Task #1 waits for receiving string and prints it.

Program code:

```
char        msg[20];
int         myrank;
int         tag = 99;
MPI_Status  status;

MPI_Comm_rank( MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
  strcpy( msg, "Hello world!");
  MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf( "%s\n", msg);
}
```

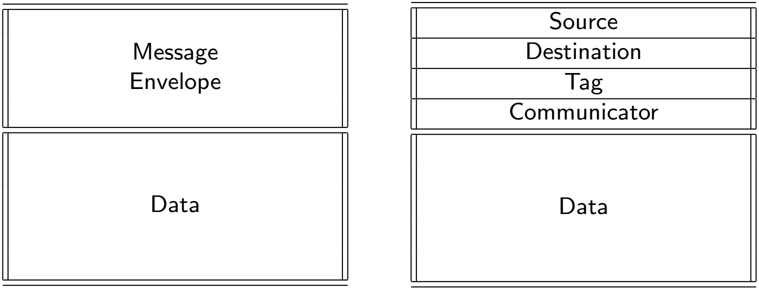
What Makes a Message ?

Message:



What Makes a Message ?

Message:



Message envelope:

- Source: sender task id
- Destination: receiver task id
- Tag: Number to distinguish different categories of messages

Standard Blocking Communication: MPI_Send

Signature:

```
int MPI_Send(
  void *buffer,              // IN : address of send buffer
  int count,                 // IN : number of entries in buffer
  MPI_Datatype datatype,     // IN : datatype of entry
  int destination            // IN : rank of destination
  int tag,                   // IN : message tag
  MPI_Comm communicator      // IN : communicator
)
```

Characteristics:

- Standard blocking send operation.
- Assembles message envelope.
- Sends message to destination.
- May return as soon as message is handed over to "system".
- May wait for corresponding receive operation.
- Buffering behaviour is implementation-dependent.
- No synchronization with receiver (guaranteed).

MPI Data Types

```
MPI datatype            C datatype
------------            ----------

MPI_CHAR                char
MPI_SIGNED_CHAR         char
MPI_UNSIGNED_CHAR       unsigned char
MPI_SHORT               short
MPI_UNSIGNED_SHORT      unsigned short
MPI_INT                 int
MPI_UNSIGNED            unsigned int
MPI_LONG                long
MPI_UNSIGNED_LONG       unsigned long
MPI_FLOAT               float
MPI_DOUBLE              double
MPI_LONG_DOUBLE         long double

MPI_BYTE
MPI_PACKED
```

Standard Blocking Communication: MPI_Recv

Signature:

```
int MPI_Recv(
  void *buffer,              // OUT : address of receive buffer
  int count,                 // IN  : maximum number of entries
  MPI_Datatype datatype,     // IN  : datatype of entry
  int source                 // IN  : rank of source
  int tag,                   // IN  : message tag
  MPI_Comm communicator,     // IN  : communicator
  MPI_Status *status         // OUT : return status
)
```

Characteristics:

- Standard blocking receive operation.
- Receives message from `source` with `tag`.
- Disassembles message envelope.
- Stores message data in buffer.
- Returns not before message is received.
- Returns additional status data structure.

Intricacies of MPI_Recv

Receiving messages from any source ?

- Use wildcard source specification `MPI_ANY_SOURCE`

Intricacies of MPI_Recv

Receiving messages from any source ?

- Use wildcard source specification `MPI_ANY_SOURCE`

Receiving messages with any tag ?

- Use wildcard tag specification `MPI_ANY_TAG`

Intricacies of MPI_Recv

Receiving messages from any source ?

- Use wildcard source specification `MPI_ANY_SOURCE`

Receiving messages with any tag ?

- Use wildcard tag specification `MPI_ANY_TAG`

Message buffer larger than message ?

- Don't worry, excess buffer fields remain untouched.

Intricacies of MPI_Recv

Receiving messages from any source ?

- Use wildcard source specification `MPI_ANY_SOURCE`

Receiving messages with any tag ?

- Use wildcard tag specification `MPI_ANY_TAG`

Message buffer larger than message ?

- Don't worry, excess buffer fields remain untouched.

Message buffer smaller than message ?

- Message is truncated, no buffer overflow.
- `MPI_Recv` returns error code `MPI_ERR_TRUNCATE`.

Status of Receive Operations

Structure containing (at least) 3 values:

- Message tag
  - used in conjunction with `MPI_ANY_TAG`
- Message source
  - used in conjunction with `MPI_ANY_SOURCE`
- Error code
  - used in conjunction with multiple receives (see later)

Status of Receive Operations

Additional information:

```
int MPI_Get_count(
  MPI_STATUS *status,        // IN  : return status of receive
  MPI_Datatype datatype,     // IN  : datatype of buffer entry
  int *count                 // OUT : number of received entries
)
```

Status of Receive Operations

Additional information:

```
int MPI_Get_count(
  MPI_STATUS *status,       // IN  : return status of receive
  MPI_Datatype datatype,    // IN  : datatype of buffer entry
  int *count                // OUT : number of received entries
)
```

Not interested in status ?

- Use `MPI_STATUS_IGNORE` as status argument !!

Type Matching

Correct message passing requires 3 type matches:

1. Sender: Variable type **must** match MPI type.
2. Transfer: MPI send type **must** match MPI receive type.
3. Receiver: MPI type **must** match variable type.

Type Matching

Correct message passing requires 3 type matches:

1. Sender: Variable type **must** match MPI type.
2. Transfer: MPI send type **must** match MPI receive type.
3. Receiver: MPI type **must** match variable type.

Strictly prohibited:

- ```
  char buf[100];
  MPI_Send( buf, 10, MPI_BYTE, dest, tag, communicator);
  ```
  ```

Type Matching

Correct message passing requires 3 type matches:

1. Sender: Variable type **must** match MPI type.
2. Transfer: MPI send type **must** match MPI receive type.
3. Receiver: MPI type **must** match variable type.

Strictly prohibited:

- ```
  char buf[100];
  MPI_Send( buf, 10, MPI_BYTE, dest, tag, communicator);
  ```
- ```
  long buf[100];
  MPI_Send( buf, 10, MPI_INT, dest, tag, communicator);
  ```

Type Matching

Correct message passing requires 3 type matches:

1. Sender: Variable type **must** match MPI type.
2. Transfer: MPI send type **must** match MPI receive type.
3. Receiver: MPI type **must** match variable type.

Strictly prohibited:

- ```
  char buf[100];
  MPI_Send( buf, 10, MPI_BYTE, dest, tag, communicator);
  ```
- ```
  long buf[100];
  MPI_Send( buf, 10, MPI_INT, dest, tag, communicator);
  ```
- ```
  MPI_Send( buf, 10, MPI_INT, 1, tag, communicator);
  MPI_Recv( buf, 40, MPI_BYTE, 0, tag, communicator, status);
  ```
  ```

Representation Conversion

Why don't we simply transmit byte vectors ?

- MPI may be used on heterogeneous systems.
- Different architectures use different encodings for same data types !
- Examples:
  - big endian vs. little endian
  - char as byte vs. char as integer
  - different floating point representations

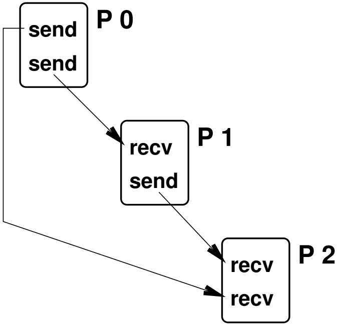
MPI implicitly cares for data conversion where necessary !

Message Ordering

The order of messages is preserved:

- for ONE source
- and ONE destination
- using ONE communicator

Is message ordering transitive ? NO !!



Programming Scalable Systems with MPI

Message Passing as a Programming Paradigm

Gentle Introduction to MPI

Point-to-point Communication

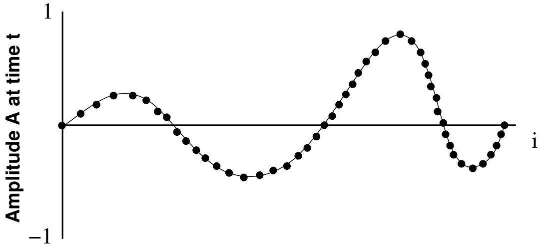
Message Passing and Domain Decomposition

Overlapping Communication with Computation

Synchronous vs Asynchronous Communication

Conclusion

Example: 1-D Wave Equation



- Update amplitude in discrete time steps.
- 1-D wave equation:

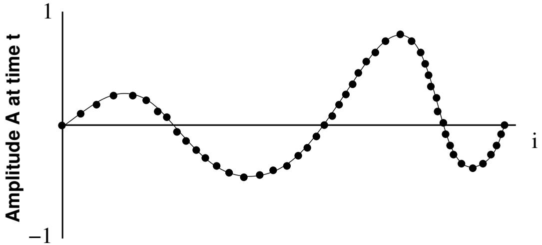
$$A_{i,t+1} \;=\; 2 \times A_{i,t} - A_{i,t-1} + c \times (A_{i-1,t} \;-\; (2 \times A_{i,t} - A_{i+1,t}))$$

- Amplitude $A_{t+1,i}$ depends on
  - Amplitude at neighbouring points
  - Amplitude at previous time steps

1-D Wave Equation: Serial Pseudo Code

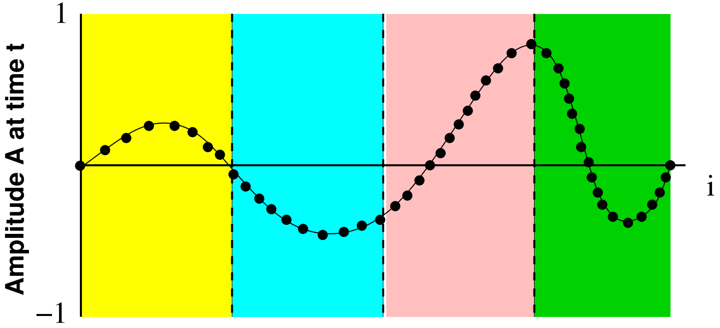
```
double cur[npoints];
double new[npoints];
double old[npoints];

initialize( cur);
initialize( old);

for t=1 to nsteps {

  for i=1 to npoints-2 {
    new[i] = 2.0 * cur[i] - old[i]
                  + c * (cur[i-1] - (2 * cur[i] - cur[i+1]));
  }

  old = cur;
  cur = new;
}

write cur to file;
```

How can we parallelise this with MPI ?



1-D Wave Equation: Parallelization Approach

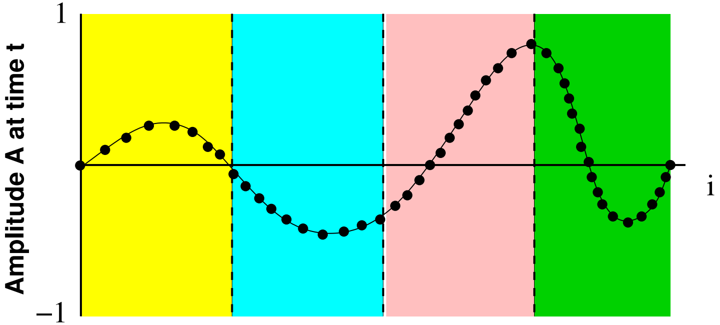
Explicit domain decomposition:



- Partition signal arrays in equally sized subarrays.
- Only store relevant fraction of signal on each node.
- Explicitly map *global indices* into *local indices*.
- Compute new signal generation locally.

1-D Wave Equation: Parallelization Approach

Explicit domain decomposition:

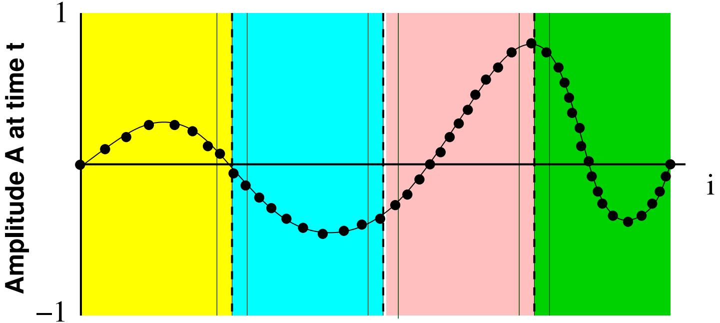


- Partition signal arrays in equally sized subarrays.
- Only store relevant fraction of signal on each node.
- Explicitly map *global indices* into *local indices*.
- Compute new signal generation locally.

But what do we do at the boundaries ?

1-D Wave Equation: Parallelization Approach

Explicit domain decomposition with halo cells:



- Add two locations for **halo cells**.
- Iterate in lock step:
  - Update halo cells.
  - Compute new signal.

1-D Wave Equation: Parallel Pseudo Code (1)

```
local_size = npoints / num_tasks();

double cur[local_size + 2];
double new[local_size + 2];
double old[local_size + 2];

left_neighbour = task_id() - 1      // Special treatment of left
right_neighbour = task_id() + 1     // and right node left out.

if (task_id() == 0) {                          // I'm the MASTER.
  for t = 1 to num_tasks()-1 {
    initialize( cur[1:local_size]) ;
    send( t, cur[1:local_size]) ;
    initialize( old[1:local_size]) ;
    send( t, old[1:local_size]) ;
  }
  initialize( cur[1:local_size]) ;
  initialize( old[1:local_size]) ;
}
else {                                         // I'm a WORKER.
  cur[1:local_size] = receive( 0);
  old[1:local_size] = receive( 0);
}
.......
```

1-D Wave Equation: Parallel Pseudo Code (2)

```
.......
for t=1 to nsteps {
  send( left_neighbour, cur[1]) ;
  cur[local_size + 1] = receive( right_neighbour);

  send( right_neighbour, cur[local_size]);
  cur[0] = receive( left_neighbour);

  for i=1 to local_size {
    new[i] = 2.0 * cur[i] - old[i]
                 + c * (cur[i-1] - (2 * cur[i] - cur[i+1]));
  }

  old = cur;
  cur = new;
}
.......
```

1-D Wave Equation: Parallel Pseudo Code (3)

```
.......

if (task_id() > 0) {                                /* I'm a WORKER. */
  send( 0, cur[1:local_size]);
}
else {                                              /* I'm the MASTER. */
  write( file, cur[1:local_size]);

  for i=1 to num_tasks() - 1 {
    cur[1:local_size] = receive( i) ;
    write( file, cur[1:local_size]);
  }
}
```

Programming Scalable Systems with MPI

Overlapping Communication with Computation

Observation:

- Communication is expensive overhead
- Communication uses network adaptor, dma controller, ...
- Computation uses cores, vector units, float units, ...

Overlapping Communication with Computation

Observation:

- Communication is expensive overhead
- Communication uses network adaptor, dma controller, ...
- Computation uses cores, vector units, float units, ...

Idea:

- Let communication happen in the background
- Run communication in parallel with computation

Overlapping Communication with Computation

Observation:

- Communication is expensive overhead
- Communication uses network adaptor, dma controller, ...
- Computation uses cores, vector units, float units, ...

Idea:

- Let communication happen in the background
- Run communication in parallel with computation

Implementation:

- Initiate message sending as soon as data is available
- Provide receive buffer as soon as old data no longer needed

1-D Wave Equation Reloaded (1)

Overlapping Communication and Computation:

```
.......
for t=1 to nsteps {
  send( left_neighbour, cur[1]) ;
  send( right_neighbour, cur[local_size]) ;

  for i=2 to local_size - 1 {
    new[i] = ... ;
  }

  cur[local_size + 1] = receive( right_neighbour) ;
  new[local_size] = ...;

  cur[0] = receive( left_neighbour) ;
  new[1] = ...;

  old = cur ;
  cur = new ;
}
.......
```

1-D Wave Equation Reloaded (1)

Overlapping Communication and Computation:

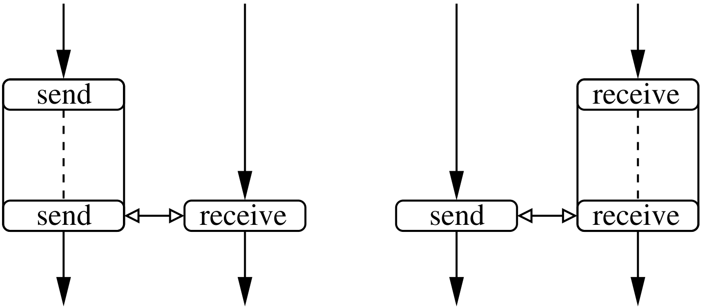
```
.......
for t=1 to nsteps {
  send( left_neighbour, cur[1]) ;
  send( right_neighbour, cur[local_size]) ;

  for i=2 to local_size - 1 {
    new[i] = ... ;
  }

  cur[local_size + 1] = receive( right_neighbour) ;
  new[local_size] = ...;

  cur[0] = receive( left_neighbour) ;
  new[1] = ...;

  old = cur ;
  cur = new ;
}
.......
```

Can we do even better ? **Homework !!**

Programming Scalable Systems with MPI

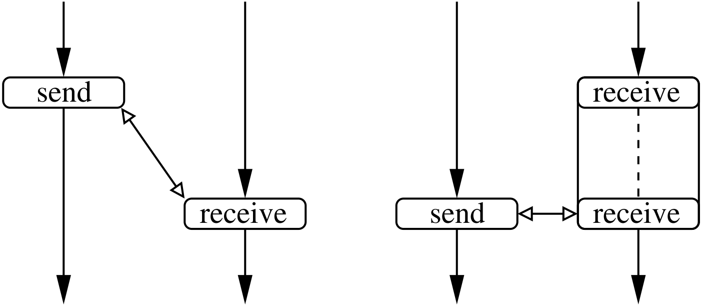
Synchronous vs Asynchronous Communication (1)

Blocking Send — Blocking Receive:



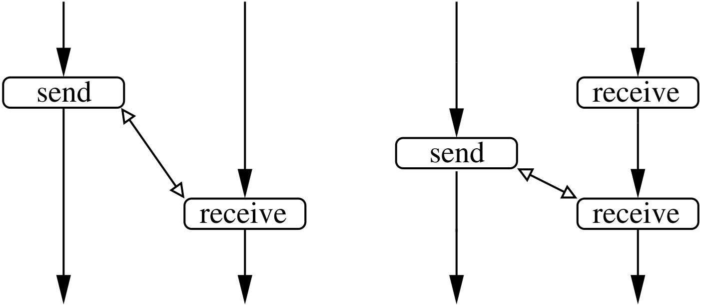
Synchronous vs Asynchronous Communication (2)

Non-Blocking Send — Blocking Receive:



Synchronous vs Asynchronous Communication (3)

Non-Blocking Send — Non-Blocking Receive:



Non-Blocking Communication

Idea:

- Split communication operation into initiation and completion.

```
                   ⎧  handle = MPI_Isend(...)
MPI_Send(...)      ⎨  ...
                   ⎩  MPI_Wait( handle, ...)

                   ⎧  handle = MPI_Irecv(...)
MPI_Recv(..)       ⎨  ...
                   ⎩  MPI_Wait( handle, ...)
```

Rationale:

- Overlap communication with computation.
- Initiate communication as early as possible.
- Complete communication as late as possible.

Non-Blocking Communication: MPI_Isend

Signature:

```
int MPI_Isend(
  void *buffer,               // IN  : address of send buffer
  int count,                  // IN  : number of entries in buffer
  MPI_Datatype datatype,      // IN  : datatype of entry
  int destination             // IN  : rank of destination
  int tag,                    // IN  : message tag
  MPI_Comm communicator,      // IN  : communicator
  MPI_Request *request        // OUT : request handle
)
```

Characteristics:

- Non-blocking send operation.
- Assembles message envelope.
- Initiates sending of message.
- Returns “immediately”.
- Does not wait for completion of sending.
- Returns request handle to identify communication operation for later inspection.

Non-Blocking Communication: MPI_Irecv

Signature:

```
int MPI_Irecv(
  void *buffer,               // OUT : address of receive buffer
  int count,                  // IN  : maximum number of entries
  MPI_Datatype datatype,      // IN  : datatype of entry
  int source                  // IN  : rank of source
  int tag,                    // IN  : message tag
  MPI_Comm communicator,      // IN  : communicator
  MPI_Request *request        // OUT : request handle
)
```

Characteristics:

- Non-blocking receive operation.
- Provides buffer for receiving message.
- Initiates receive operation.
- Does not wait for message.
- Returns "immediately".
- Returns request handle to identify communication operation for later inspection.

Non-Blocking Communication: MPI_Wait

Signature:

```
int MPI_Wait( MPI_Request *request,    \\ INOUT : request handle
              MPI_Status *status       \\ OUT   : return status
)
```

Characteristics:

- Finishes non-blocking send or receive operation.
- Returns not before communication is completed.
- Sets request handle to `MPI_REQUEST_NULL`.
- Returns additional status data structure.

Non-Blocking Communication: MPI_Test

Signature:

```
int
MPI_Test(
  MPI_Request *request,   \\ INOUT : request handle
  int *flag               \\ OUT   : true iff operation completed
  MPI_Status *status      \\ OUT   : return status
)
```

Characteristics:

- Checks status of non-blocking send or receive operation.
- Returns immediately.
- Flag indicates completion status of operation.
- If operation is completed, sets request handle to `MPI_REQUEST_NULL`.
- If operation is completed, returns additional status data structure.
- If operation is still pending, `MPI_Test` does nothing.

1-D Wave Equation Reloaded Once More

How could the wave equation benefit ?

Homework !!

Programming Scalable Systems with MPI

Message Passing as a Programming Paradigm

Gentle Introduction to MPI

Point-to-point Communication

Message Passing and Domain Decomposition

Overlapping Communication with Computation

Synchronous vs Asynchronous Communication

Conclusion

MPI and Shared Memory Multi-Core Nodes

History:

- MPI invented in uni-core era
- Networked large-scale SMPs uncommon
  (poor price/performance ratio)

Options today:

- Run multiple MPI processes per node
- Implementation trick: communication via shared memory
- Combine MPI with OpenMP / PThreads
- Future versions of MPI will have dedicated SMP support

Summary and Conclusion

Global view programming with Pthreads or OpenMP:

- Multiple concurrent execution threads **within** process
- Concurrent access to shared data
- Race conditions
- Deadlocks

Local view programming with MPI:

- Multiple concurrent processes
- Large data structures require explicit splitting
- Array index mapping between global and local view needed
- Data marshalling / unmarshalling needed
- Deadlocks

The End: Questions ?