CME 213

Spring 2012-2013

Eric Darve

MPI

Parallel programming using Message Passing

Flynn's taxonomy

We have seen the following different types of parallelism:

SIMD: single instruction multiple data

- All processing units execute the same instruction at any given clock cycle
- Each processing unit can operate on a different data element

This applies for example to GPU processors. The thread index is used to determine which data the thread operates with.

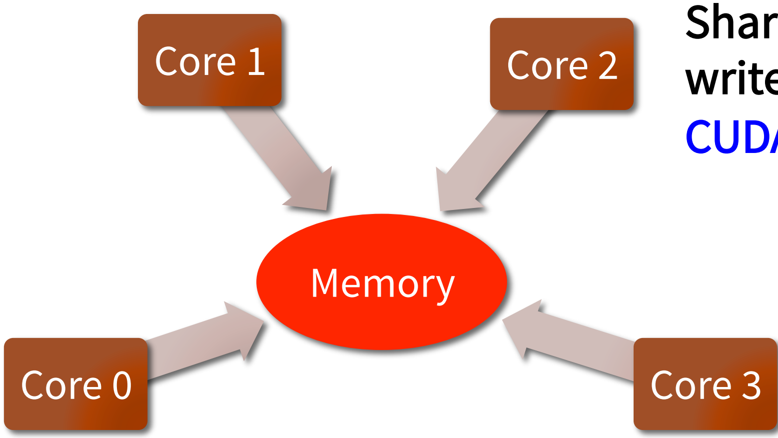
MIMD: multiple instruction, multiple data

- Every processor executes a different instruction stream
- Every processor works with a different data stream

This applies to Pthreads for example. Threads can execute different routines.

MPI follows this model.

Parallel computer memory architecture

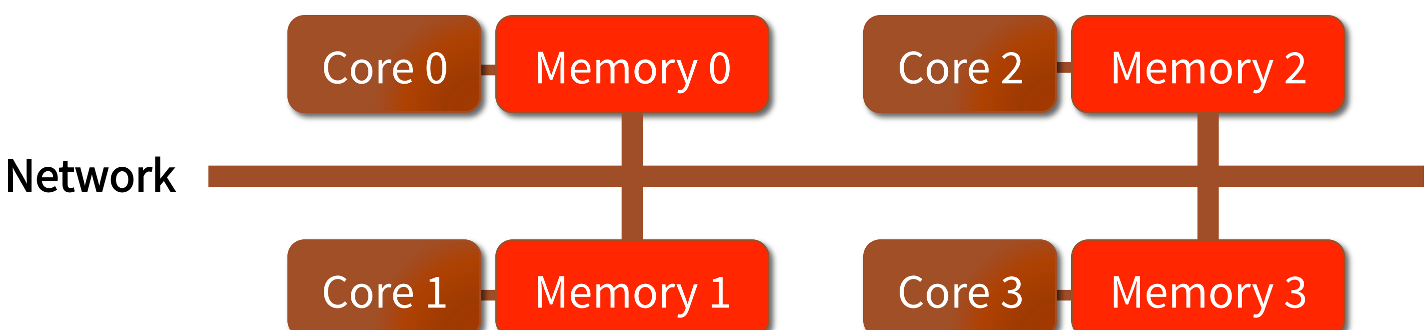


Shared memory: all cores can read and write to the same memory space
CUDA, Pthreads, OpenMP

Distributed memory:

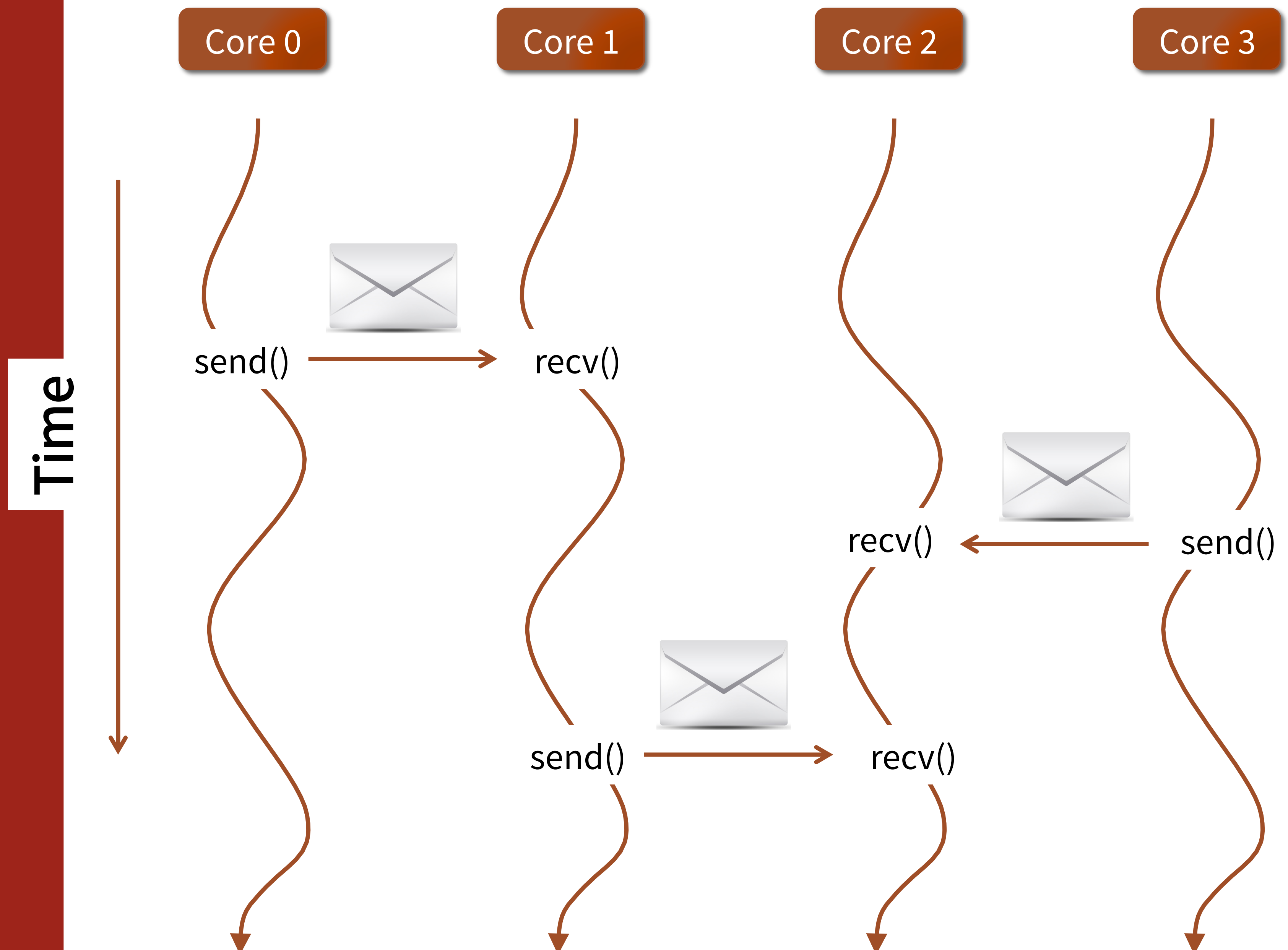
- cores read and write to the different memory spaces
- If a core needs data in some other memory space, explicit communication is required

MPI



MPI: MESSAGE PASSING INTERFACE

- All processes run the same program.
- Processes are assigned a rank.
- Based on the rank, processes perform calculations on different data.
- Processes communicate by sending and receiving messages.
- Message passing:
  - Data transfer requires cooperative operations to be performed by each process.
  - For example, a send operation must have a matching receive operation.


Core 0
Core 1
Core 2
Core 3
Time
send()
recv()
recv()
send()
send()
recv()

Our first MPI program

MPI IMPLEMENTATIONS

- MVAPICH: mvapich.cse.ohio-state.edu
- MPICH: www-unix.mcs.anl.gov/mpi/mpich2
- LAM/MPI: www.lam-mpi. org
- OpenMPI: www.open-mpi.org

- Download your favorite implementation and install it!
- You can test MPI using a multicore computer.
- Each process can then run on its own core.

COMPILATION AND RUNNING

Header file:

`#include “mpi.h”`

Compilation:

C: **`mpicc`**, …

C++:   **`mpiCC`**, …

Run:

```
mpiexec -n 4 program arguments
```

or

```
mpirun -np 4 program arguments
```

- This will run the code **`program`** using 4 processes of the cluster.
- All nodes run the same program.
- The processes may be running on different cores of the same node.

EXAMPLE

- Take two processes and have one process send an integer to the other process.
- See example code.

SEND

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

- `smessage` buffer which contains the data elements to be sent
- count number of elements to be sent
- `datatype` data type of entries
- `dest` rank of the target process
- `tag` message tag which can be used by the receiver to distinguish between different messages from the same sender
- `comm` communicator used for the communication (more on this later)

RECV

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

Same as before. New argument:

- `status` data structure that contains information about the message that was received

MPI DATA TYPES

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

How does it work?

- Each `Send` must be matched with a corresponding `Recv`.
- Order: messages are delivered in the order in which they have been sent.
  - If a sender sends two messages of the same type one after another to the same receiver, the MPI runtime system ensures that the first message sent will always be received first.

Point-to-point communication

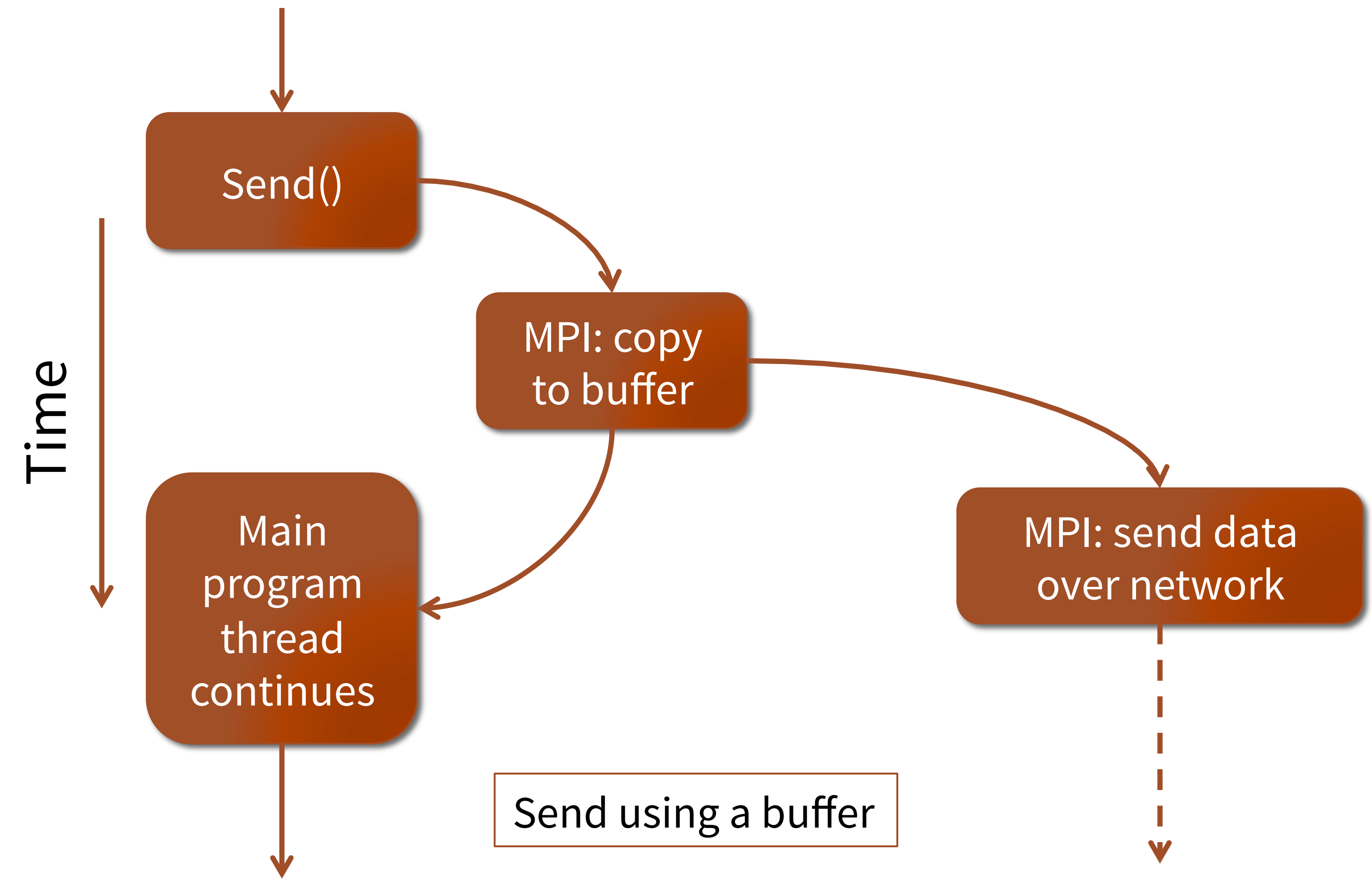
Stanford University

Point-to-point communication

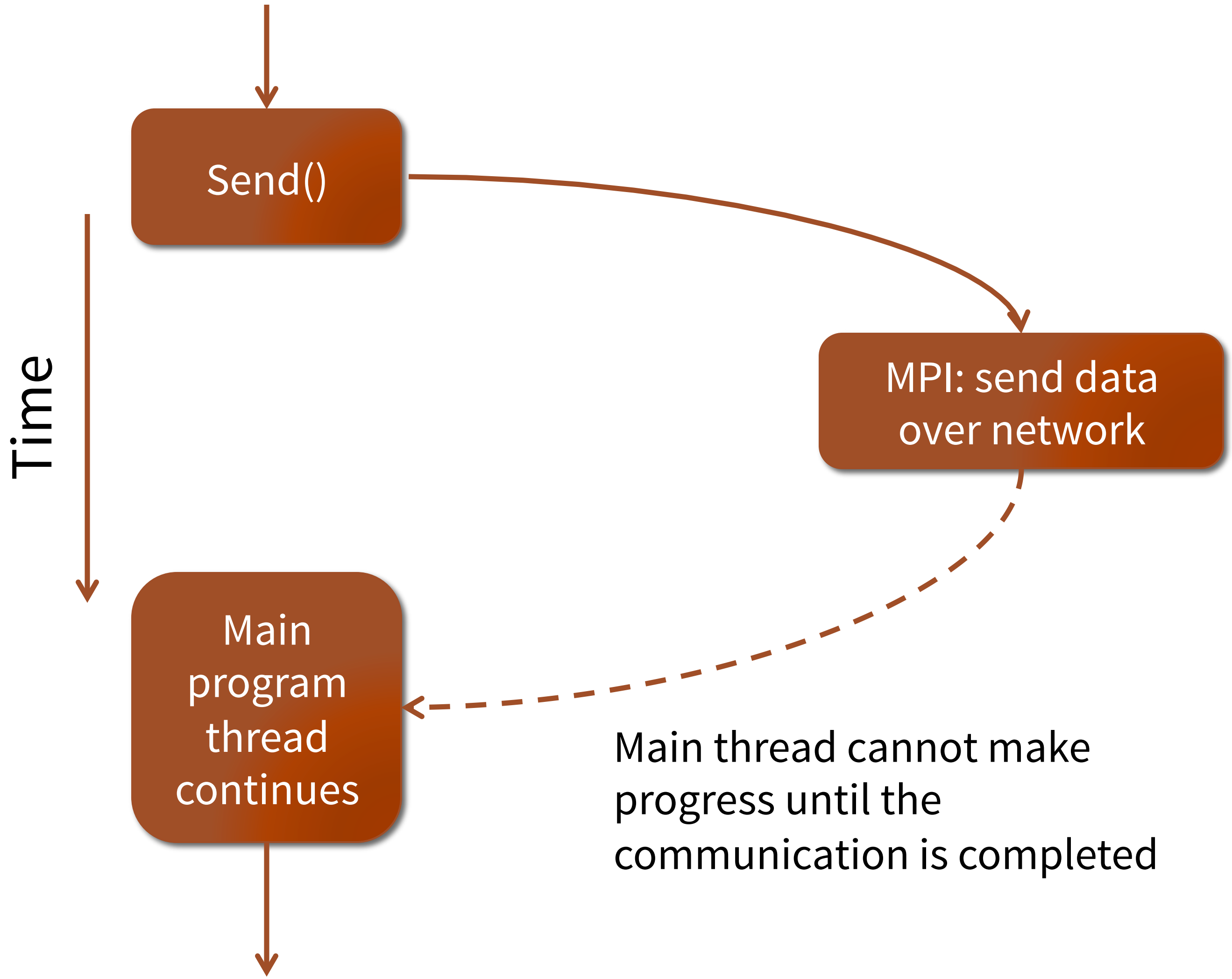
- There are a few technical details to understand regarding communication.
- This is important to understand whether a deadlock may occur in your program or not.
- Two key concepts:
    - Blocking/non-blocking
    - Synchronous/asynchronous

Two ways to communicate – 1) using an MPI system buffer

To optimize the communication the MPI library uses two different strategies for communication: buffered and non-buffered.



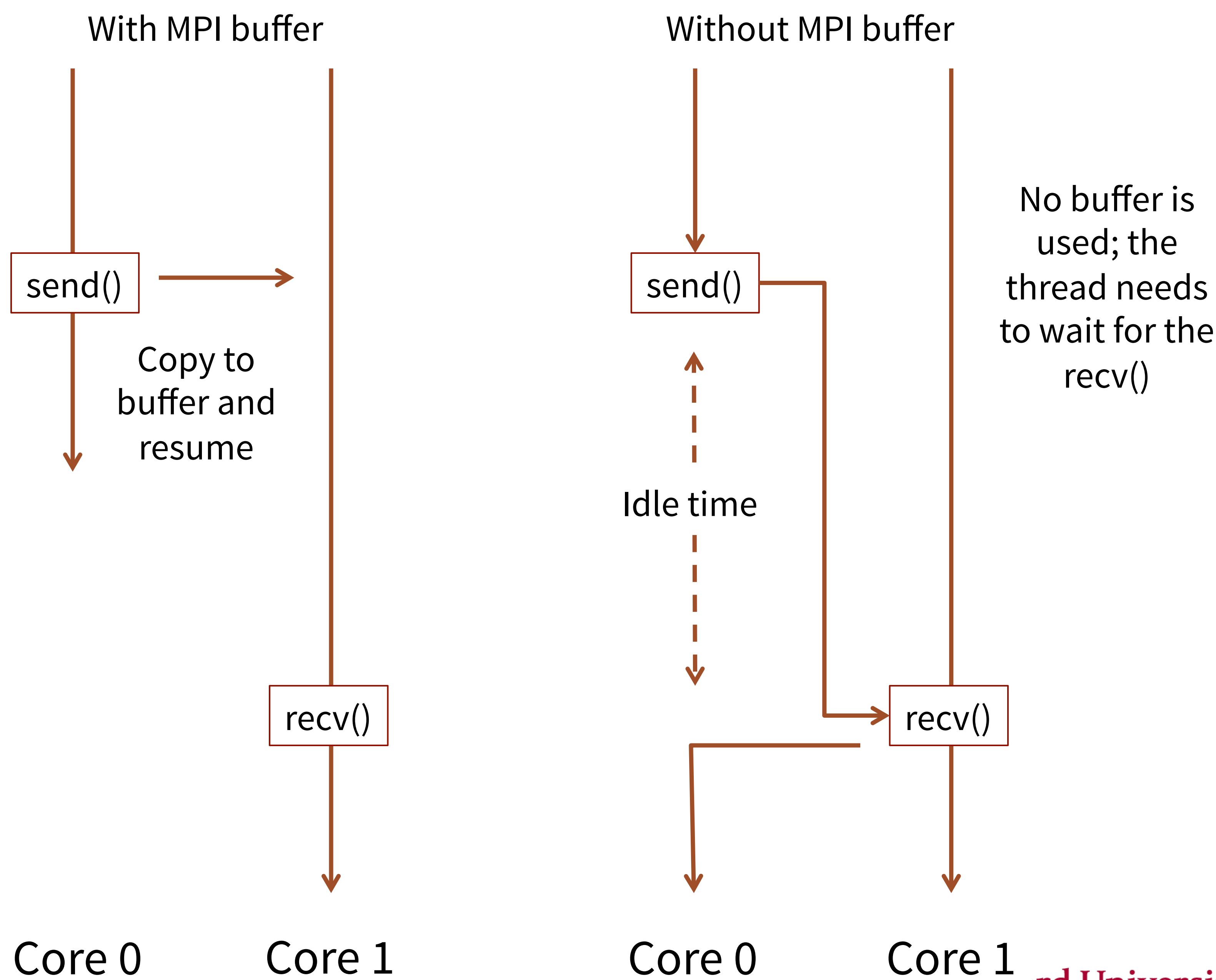
2) WITHOUT A BUFFER



Main thread cannot make progress until the communication is completed

WHAT IS THE DIFFERENCE?

- Send and Recv are blocking operations:
  - The call does not return until the resources become available again
  - Send: data in buffer can be changed
  - Recv: data in buffer is available and can be used
- `Send` – If MPI uses a separate system buffer, the data in `smessage` (user buffer space) is copied (fast); then the main thread resumes.
- If MPI does not use a separate system buffer, the main thread must wait until the communication over the network is complete.
- This is similar for `Recv`. If communication happens before the call, the data is stored in an MPI system buffer and then simply copied into the user provided `rmessage` when recv() is called.
- Note: the user cannot decide whether a buffer is used or not; the MPI library makes that decision based on the resources available and other factors.


With MPI buffer
Without MPI buffer
send()
Copy to buffer and resume
recv()
send()
Idle time
recv()
No buffer is used; the thread needs to wait for the recv()
Core 0
Core 1
Core 0
Core 1