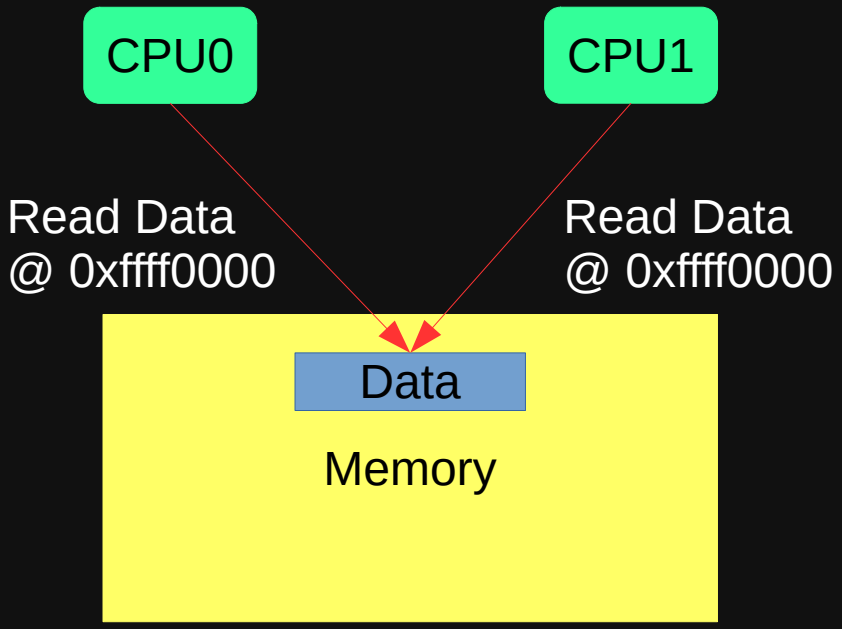
Distributed Memory Programming with Messages

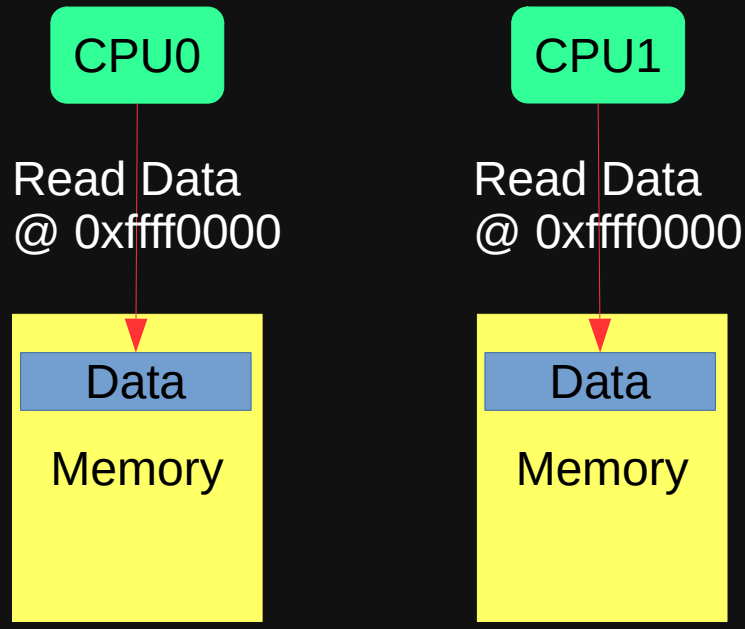
Wei Wang

Shared-Memory VS Distributed-Memory

- Shared-Memory: all processors access the same chunk of memory with the same address



- Distributed-memory: processors access different chunks of memory with the same address



Distributed Memory Parallel Programming

- More generally, distributed memory systems include processes on different machines
- Each process has its own exclusive address space
- All data must be explicitly partitioned and placed
- All interactions (communications) must be explicitly expressed
  - Send or receive
  - Read or write

Message Passing Based Parallel Programming

- Message passing is a common communication mechanism for distributed memory parallel programming.
- Data are sent and received through messages
- All interactions are two-sided
  - Process that has the data sends it
  - Process that wants the data receives it
- Strengths:
  - Simple performance model: communication costs are explicit
  - Portable
- Weakness
  - Two-sided model can be awkward to program

Pros and Cons of Message Passing

- Advantages
  - universality
    - works well on machines with fast or slow network
  - expressibility
    - useful and complete model for expressing parallel algorithms
  - lack of data races
    - data races are a pervasive problem for shared memory programs*
  - performance
    - yields high performance by co-locating data with computation
- Disadvantages
  - managing partitioned address spaces is a hassle
  - two-sided communication is somewhat awkward to write
  - debugging multiple processing is awkward

* MPI is not devoid of race conditions, but they can only occur when non-blocking operations are used

Message Passing Flavors: Blocking, Unbuffered

- Definition
  - send won't return until its data has been received by the receiver
  - receive won't return until data has arrived from the sender
  - no extra copies made of message data
- Advantage:
  - simple to use
- Disadvantages
  - send and recv may idle, and idling hurts performance
  - deadlock is possible since send operations block

```
Process 0

send x to process 1

receive y from process 1
```

Both processes try to send, and both are indefinitely blocked, b/c no one receives

```
Process 1

send y to process 0

receive x from process 0
```

Message Passing Flavors: Blocking, Buffered

- Definition
  - send won't return until its data is transferred to a buffer
    - Buffer may be at the sender side: may return after data copied into a buffer at sender
    - Buffer may be at the receiver side: may return after data copied into a buffer at receiver
  - receive won't return until the data has arrived
- Advantages
  - simple to use
  - avoids deadlock caused by send
- Disadvantages
  - receive may idle waiting for a send, and idling hurts performance
  - deadlock still possible since receive operations block

```
Process 0

receive y from process 1

send x to process 1
```

Both processes try to receive, and both are indefinitely blocked, b/c no one sends

```
Process 1

receive x from process 0

send y to process 0
```

Buffered Blocking Message Passing

- Buffer sizes can have significant impact on performance

```
Process 0

for (i = 0; i < N; i++){
  produce_data(&a);
  send a to proc 1;
}
```

```
Process 1

for (i = 0; i < N; i++){
  receive a from proc 0;
  consume_data(&a)
}
```

With large buffers, the sender can continuously send data to the receiver by copying multiple data into the buffer. The sender does not have to stop to wait for the receiver to retrieve the data from the buffer.

If the buffer is too smaller, the sender will be blocked if the buffer is full and will be forced to wait for the receiver to retrieve data and release buffer space.

Message Passing Flavors: Non-Blocking

- Definition
  - send and receive functions return before the operation is completely finished
    - sender: data can be overwritten before it is sent
    - receiver: can read data out of buffer before it is received
  - ensuring proper usage is the programmer's responsibility
  - status check operation to ascertain completion
- Advantages
  - tolerate asynchrony
  - overlap communication with computation
- Disadvantage
  - programming complexity

Process 0

```
start_send x to proc 1
Start_recv y from proc 1

… do some work …

end_send x to proc 1
end_recv y from proc 1
```

Needs to be careful with the code between start and finish, b/c send and recv not actually finish

Process 1

```
start_send y to proc 0
Start_recv x from proc 0

… do some work …

end_send y to proc 0
end_recv x from proc 0
```