CS 3211
Parallel Programming Models

Abhik Roychoudhury
National University of Singapore

Gives basic concepts of parallel programming before moving to MPI

Parallel Programming Models

- Message-passing programming paradigm
  - Oldest and most widely used approach for programming parallel computers
  - Imposes minimal requirements on the underlying hardware
- Shared-memory programming paradigm
  - Increasingly popular with the emergence of shared memory multiprocessors on chip
- **We will study message passing programming paradigm using Message Passing Interface (MPI)**

Message Passing Programming

- Partitioned address space
  - Logical view consists of *p* processes, each with its own exclusive address space
  - Each data element must belong to one of the partitioned address space --- explicit data partitioning
    - Programming difficulty but better locality of access
  - All interactions require cooperation of both the process that has the data and the process that wants the data
    - Programming difficulty but forces programmers to minimize interactions
  - Natural fit for clustered workstations
- Supports only explicit parallelization

Structure of message passing programs

- Asynchronous
  - All concurrent tasks execute asynchronously
  - Programs are hard to reason about due to non-deterministic behavior.
- Loosely synchronous
  - Tasks synchronize to perform interactions; between interactions tasks execute completely asynchronously
- Single Program Multiple Data (SPMD)
  - Code executed by different processes are identical

So far

- We have discussed 2 kinds of message passing
  - Synchronous message passing
    - Handshake between sender and receiver processes
    - Send is blocking – until there is a matching receive.
    - Similarly receive also must be blocking.
  - Asynchronous message passing
    - Send is non-blocking, data put into a buffer.
    - Receive still occurs after the send, by taking data from buffer.
- In actual implementations
  - Things can get more complicated.
  - Need to distinguish between user and system/NW space in each process.
  - **Welcome to parallelism …**

User and Network space

- ! data
  - Copy data from user space into the network/system space.
  - Actual communication can take place later.
- ?data
  - Copy data from network space into user space at receiver end.
  - Again actual communication take place later, fool the system to believe that receive has taken place.
  - This can be safe, as long as receiver process is not manipulating the variables where the data supposed to have been received would have been stored.
  - Need to check for actual receives from time to time.

In actual parallel implementations

- There are always buffers in network space.
  - Buffered send => Buffers in user and network space
  - Non-buffered send => Buffer only in network space
    - More discussion later when we discuss MPI.
- Message transmission is via network packets.
  - … travelling over the network.
- Implementation of message send
  - Transfer data from user space to network buffers typically by DMA. This will then travel over the network.
- Implementation of message receive
  - Transfer of data from network buffers to user space back. This data has by now travelled over the network.

PAUSE please

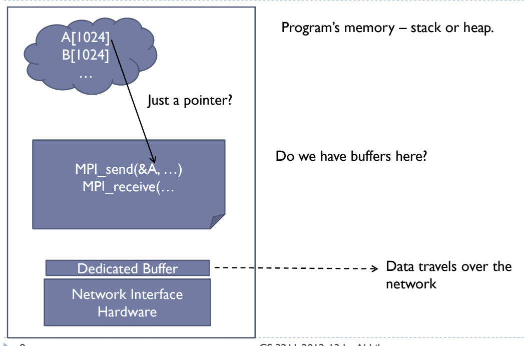
We now discuss for 4 comm. schemes:
blocking / non-blocking, buffered / non-buffered

All the Sequence Diagrams shown for the 4 communication schemes are a schematic only.
Exact implementation may differ for different versions of MPI.
The key issue is **not** the exact implementation of these schemes.
The key issue is to understand the actual **software system layers** in each process – be it the sender, or receiver.

Only by getting an appreciation of the different layers in each process
-Program memory
-MPI libraries
-Network interface

you can get an actual appreciation of how message passing communication takes place in a parallel system.

The actual system stack



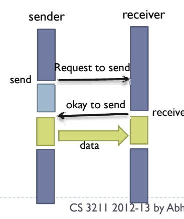
Building Blocks: Send and Receive

- Interactions are accomplished by sending and receiving messages
  - `send (void *sendbuf, int nelems, int destination)`
  - `receive(void *recvbuf, int nelems, int source)`

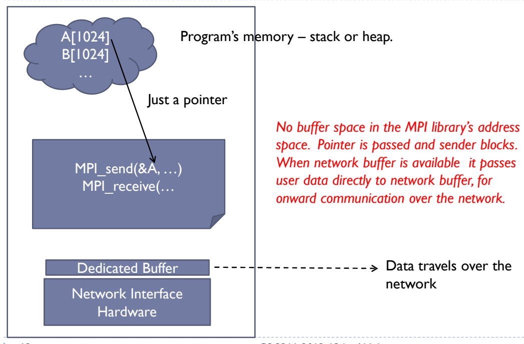
```
 P0                          P1

a = 100;                receive(&a, 1, 0);
send (&a, 1, 1);        printf("%d\n", a);
a = 0;
```

Blocking non-buffered send/receive

- Send operation does not return until the matching receive has been encountered at the receiving process
- Involves handshake between sender and receiver
- Drawback: Idling overhead



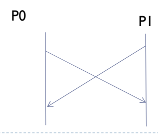
Blocking non-buffered


A[1024]
B[1024]
…
Program's memory – stack or heap.
Just a pointer
MPI_send(&A, …)
MPI_receive(…
Dedicated Buffer
Network Interface Hardware
Data travels over the network

No buffer space in the MPI library's address space. Pointer is passed and sender blocks. When network buffer is available it passes user data directly to network buffer, for onward communication over the network.

Deadlock in Blocking Non-buffered send/receive

```
 P0                      P1

send (&a, 1, 1);        send(&a, 1, 0);
receive (&b, 1, 1);     receive(&b, 1, 0);
```

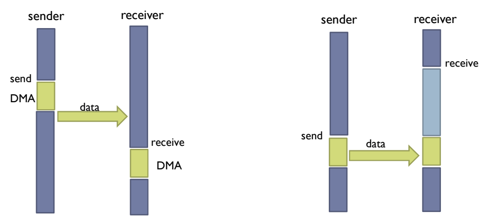


If only the sends were non-blocking !!

Blocking buffered send/receive

- Sender has a buffer pre-allocated for messages
- On encountering a send operation, the sender simply copies the data into the designated buffer and returns
  - Data goes from user space to network space
  - Actual communication can take place later
- At receiving end, data is again copied into a buffer
  - When receiving process encounter a receive operation, it checks if data is available in the buffer and copies it into target location
- Removes idling overhead at sender
- Drawback: Buffer overflow

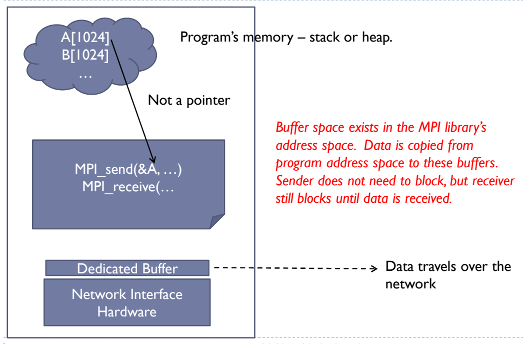
Blocking buffered send/receive



This will lead to non-blocking sends, but blocking receives.

Receive operation cannot go ahead until the data is actually received.

Blocking buffered


A[1024]
B[1024]
…
Program's memory – stack or heap.
Not a pointer
MPI_send(&A, …)
MPI_receive(…
Dedicated Buffer
Network Interface Hardware
Data travels over the network

Buffer space exists in the MPI library's address space. Data is copied from program address space to these buffers. Sender does not need to block, but receiver still blocks until data is received.

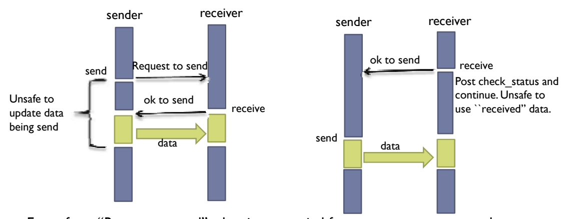
Deadlock in buffered send/receive

```
 P0                      P1

receive (&a, 1, 1);     receive(&a, 1, 0);
send (&b, 1, 1);        send (&b, 1, 0);
```

Non-blocking message passing

- Returns from send or receive operation before it is semantically safe to do so
- User must not alter the data potentially participating in a communication operation
- Generally accompanied by a check-status operation which indicates if the communication has completed
- Non-blocking operations can again be buffered or non-buffered

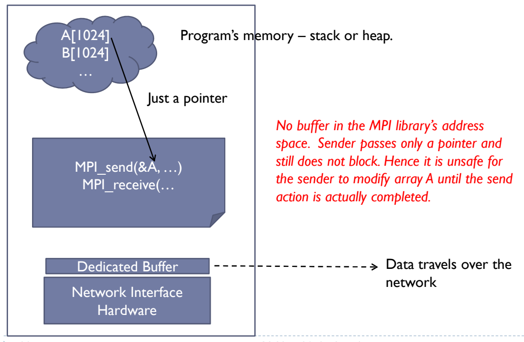
Non-blocking non-buffered


sender
receiver
send
Request to send
ok to send
receive
data
Unsafe to update data being send
sender
receiver
ok to send
receive
Post check_status and continue. Unsafe to use ``received'' data.
send
data

Even after "Request to send" , data is not copied from user space to network space. Hence it should not be updated meanwhile.

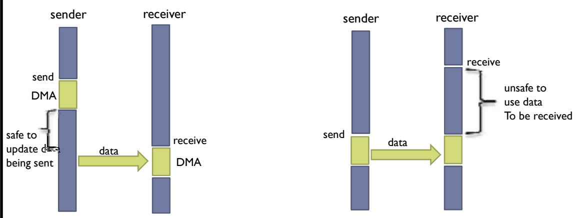
Send is non-blocking.
Receive is also non-blocking.
Receiver meanwhile checks whether data has arrived via check_status

Non-blocking non-buffered


A[1024]
B[1024]
…
Program's memory – stack or heap.
Just a pointer
MPI_send(&A, …)
MPI_receive(…
Dedicated Buffer
Network Interface Hardware
Data travels over the network

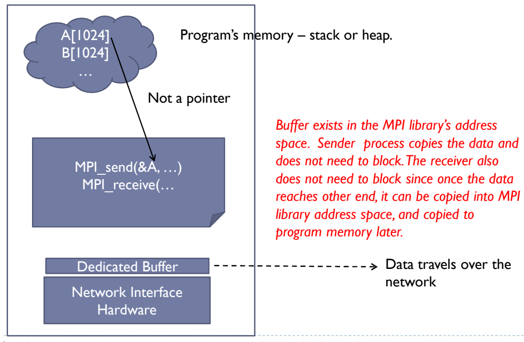
No buffer in the MPI library's address space. Sender passes only a pointer and still does not block. Hence it is unsafe for the sender to modify array A until the send action is actually completed.

Non-Blocking buffered send/receive



Both send and receive are non-blocking.
Difference with previous scheme --- we now have buffers.
Sender can modify the data being sent since it has already been copied to buffers.

Non-blocking non-buffered

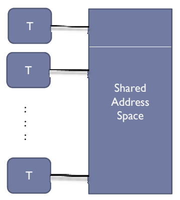

A[1024]
B[1024]
…
Program's memory – stack or heap.
Not a pointer
MPI_send(&A, …)
MPI_receive(…
Dedicated Buffer
Network Interface Hardware
Data travels over the network

Buffer exists in the MPI library's address space. Sender process copies the data and does not need to block. The receiver also does not need to block since once the data reaches other end, it can be copied into MPI library address space, and copied to program memory later.

Shared memory programming model

Alternative to message passing based programming that we will be learning.

- All memory in the logical machine model of a thread is globally accessible to every thread in the system



Threads

- A thread is a single stream of control in the flow of a program

```
for (row = 0; row < n; row++)
  for (col = 0; col < n; col++)
     c[row][col] = create_thread(dotproduct(get_row(a,row),
                                          get_col(b,col)));
```

Why threads?

- Software portability: Threaded application can be developed on serial machines and run on parallel machines without any change
- Latency hiding: While one thread is waiting for communication operation, another thread can utilize the processor
- Scheduling and load balancing: Programmer must express concurrency in a way that minimizes communication and idling
- Ease of programming: Shared memory programs are much easier to write than message-passing programs. **Why?**

Synchronization Primitives

- Much effort on synchronizing concurrent threads with respect to their data accesses or scheduling
- When multiple threads attempt to manipulate the same data item, the result can be incoherent if proper care is not taken to synchronize them

```
/* each thread tries to update variable best_cost as follows */
if (my_cost < best_cost)
        best_cost = my_cost;

/* Initial value of best_cost = 100;
   my_cost = 50 and 75 in threads T1 and T2 */
```

Mutex lock

- Support for implementing critical sections and atomic operations
- To access shared data, a thread must first try to acquire mutex
- At any point in time, only one thread can lock a mutex
- If the mutex is already locked, the thread trying to acquire the lock is blocked
- When a thread is done accessing shared data, it should release the mutex

Barrier

- A barrier call is used to hold a thread until all the other threads participating in the barrier have reached the barrier

Sequential Code

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];

sum = 0;
for (i=0; i<8; i++)
    if (a[i] > 0)
        sum = sum + a[i];
print sum;
```

Message passing code

```
id = getmyid();
local_iter = 4;
start_iter = id * load_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4…7]);
else
  recv_msg (P0, b[4..7], c[4..7];

 for (i=start_iter; i<end_iter; i++)
     a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0)
           local_sum = local_sum + a[i];
if (id == 0) {
     recv_msg (P1, &local_sum1);
     sum = local_sum + local_sum1;
     print sum;
}
else
     send_msg (P0, local_sum);
```

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];

sum = 0;
for (i=0; i<8; i++)
    if (a[i] > 0)
        sum = sum + a[i];
print sum;
```

Shared memory code

```
begin parallel     // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4;
shared double sum = 0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
    a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0) {
          lock (mylock);
          sum = sum + a[i];
          unlock (mylock);
      }
barrier;
end parallel // kill the child thread
print sum;
```

Shared memory vs. message passing

Aspects	Shared Memory	Message Passing
• Communication	• Implicit (load/store)	• explicit (via msg)
• Synchronization	• Explicit	• Implicit (via msg)
• Hardware Support	• Typically required	• None
• Development effort	• Lower	• Higher
• Tuning effort	• higher	• lower

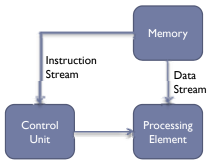
Flynn's Taxonomy of Parallel Computers

A very brief look (5 minutes!) at parallel architectures, before proceeding with MPI

		Number of Data Streams	
		Single	Multiple
Number of instruction streams	Single	SISD	SIMD
	Multiple	MISD	MIMD

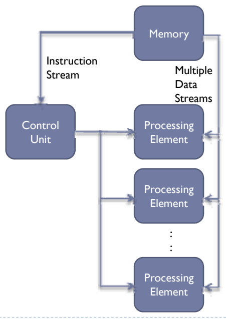
SISD: Single Instruction Single Data

- SISD is not considered as a parallel architecture
- SISD exploits parallelism at the instruction level
- Pipelined, Superscalar, and VLIW architectures are examples of SISD architecture



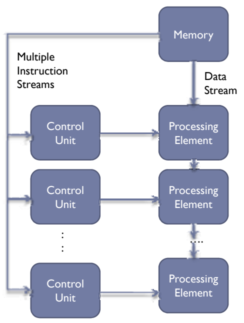
SIMD: Single Instruction Multiple Data

- A single instruction operates on multiple data to exploit data parallelism
- Vector processors and GPUs are excellent examples of SIMD architecture



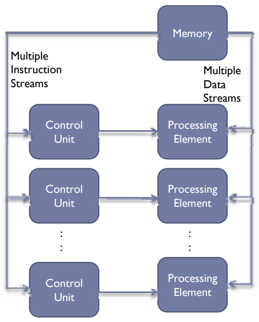
MISD: Multiple Instruction Single Data

- Multiple processing elements execute from different instruction streams and data is passed from one processing element to the next
- Example: Systolic array such as CMU iWrap
- Data passing restriction is quite severe --- hard to generalize



MIMD: Multiple Instruction Multiple Data

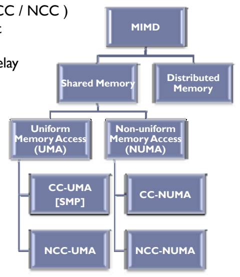
- Most flexible architecture
- Used in most parallel computers today



MIMD Classification

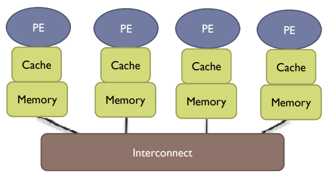
Processor has cache (CC / NCC)
CC == Cache coherent

Processor to memory delay
(UMA / NUMA)



Distributed Memory MIMD

- Processing elements (PE) work independently
- Interaction is via message passing
- PEs cannot access the memory of another PE directly

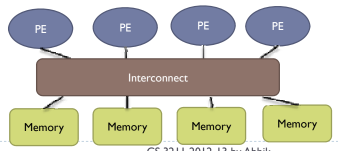


Distributed Memory MIMD (contd.)

- Advantages
  - Processors mostly work with local memory
    - less contention and highly scalable
  - As communication is via message passing, sophisticated synchronization techniques are not required
- Disadvantages
  - Load balancing (partition of code and data) responsibility is on the programmer
  - Message-passing-based synchronization can lead to deadlocks; programmer's responsibility
  - Performance overhead due to physical data copy

Shared Memory MIMD

- Multiple processors with a logically shared memory
- The memory modules define a global address space that is shared among the processors
- Any processor can access any memory module
- Communication though memory loads and stores
- Logically shared memory can be physically distributed



Shared Memory MIMD (contd.)

- Advantages:
  - No need to partition code or data
  - No need to physically move data among processors
    - communication is efficient
- Disadvantages:
  - Special synchronization constructs are required
  - Lack of scalability due to contention