High Performance Computing

Lecture 6:
Programming Using the Message-Passing Paradigm

Agenda

- Message Passing Programming Model
- Blocking communication
  - Non-Buffered
  - Buffered
- Non-Blocking communication
- Introducing MPI
Parallel Programming Models

- Programming models provide support for expressing concurrency and synchronization:
  - Shared Address space
  - Message passing
Message-passing programming model

- Logical view of a machine supporting the message passing paradigm:
  - $P$ processors.
  - Exclusive (distributed) address space.

Principles

- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed $\rightarrow$ adds complexity to programming.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data $\rightarrow$ unnatural programs.
- Parallelism is coded explicitly by the programmer:
  - Analysis,
  - Decomposition,
  - Concurrency,
  - Scalability..........................(intellectually demanding)
Principles

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the *asynchronous* paradigm, all concurrent tasks execute asynchronously.
- In the *loosely synchronous* model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.

Programming model

- Most message-passing programs are written using the *single program multiple data* (SPMD) model.
  - "*One program is executed on all processors*"
- Each processor is assigned a unique identifier, it is possible to have the processors do different work, even though they are running the same program.
Programming model

- Program gets started with all processes
- Each process executes one and the same program code
- Control flow instructions separate code to be executed by a certain process.
- All processes terminate at the end of the program

The Building Blocks: Send/Receive

- Interactions are accomplished by sending and receiving messages.
- The basic operations in the message-passing programming paradigm are send and receive.
Send and Receive Operations

- The prototypes of these operations are as follows:
  
  send(void *sendbuf, int nelems, int dest)
  
  receive(void *recvbuf, int nelems, int source)

Example

```
P0
a = 100;
send(&a, 1, 1);
printf("%d\n", a);
a = 0;

P1
receive(&a, 1, 0)
```

What happens??

How it should be implemented??

various implementation details and message passing protocols to help in ensuring the semantics of the send and receive operations
Send and Receive Operations

- Most message passing platforms have additional hardware support for sending and receiving messages
  - DMA (direct memory access) and
    - allows copying of data from one memory location to another (e.g., communication buffers) without CPU support.
  - asynchronous message transfer using network interface hardware:
    - allow the transfer of messages from buffer memory to desired location without CPU intervention.

- Two modes of Communication between processes:
  - Blocking
  - Non-Blocking

blocking vs. Non-Blocking Communication

After the break!!
The program will not return from the subroutine call until the copy to/from the system buffer has finished and is ready for use by the program.

A blocking send can be:

- **synchronous**: which means there is handshaking occurring with the receive task to confirm a safe send.
- **asynchronous**: if a system buffer is used to hold the data for eventual delivery to the receive.

Handshake for a **synchronous** blocking non-buffered send/receive operation. In cases (b) and (c) sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.
A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.

Consider the following code segments:

- What happens??
- How to correct??

Blocking Message Passing

```c
P0  send(&a, 1, 1);
    receive(&b, 1, 1);

P1  send(&b, 1, 0);
    receive(&a, 1, 0);
```

Infinite wait → deadlock

Buffered Blocking Message Passing

- A solution to the idling and deadlock problem is to rely on buffers at the sending and receiving ends:
  - The **sender** copies the data into the designated buffer and returns after the copy operation has been completed.
  - The **receiving** end buffers the data as well.
- Buffering trades off idling overhead for buffer copying overhead.
**Application buffer:**
- User managed address space (program variables).

**System buffer:**
- Area is reserved to hold data in transit
- Managed entirely by the MPI library.
  Allows receive operations to be asynchronous.

---

**Buffering**

**Blocking buffered transfer protocols**

(a) In the presence of communication hardware with buffers at send and receive ends;

(b) In the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.
**Bounded buffer sizes impact on performance**

if consumer was much slower than producer $\rightarrow$ buffer overflow

How to fix that??

---

**P0**

```c
for (i = 0; i<1000; i++)
{
    produce_data(&a);
    send(&a, 1, 1);
}
```

**P1**

```c
for (i = 0; i<1000; i++)
{
    receive(&a, 1, 0);
    consume_data(&a);
}
```

---

**Deadlocks?**

```c
P0
receive(&a, 1, 1);
send(&b, 1, 1);
```

```c
P1
receive(&b, 1, 0);
send(&a, 1, 0);
```
In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered).

How to ensure semantic correctness and provide a fast send/receive operation with little overhead?
- non-blocking protocols return from the send or receive operation before it is semantically safe to do so.

Non-blocking non-buffered send/receive
(a) In absence of communication hardware;
(b) In presence of communication hardware.
Non-Blocking Message Passing

- Non-blocking operations can also be used with a buffered protocol.
  - the sender initiates a DMA operation and returns immediately.
  - The data becomes safe the moment the DMA operation has been completed.
  - At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location.
- **Advantage:** reducing the time during which the data is unsafe.
Overview

- Early generation commercial parallel computers were based on the message-passing architecture
  - lower cost relative to shared-address-space.
  - used message-passing programming paradigm
  - The development of different libraries from different vendors:
    - Syntactic & serious semantic
    - Standardization needed
What is MPI?

- **MPI** = Message Passing Interface
- MPI is a specification for the developers and users of message passing libraries. By itself, it is **NOT** a library – but rather the specification of what such a library should be.

History and Evolution

- MPI resulted from the efforts of numerous individuals and groups (parallel computer vendors, software writers, academia and application scientists).
- **1980s – early 1990s**: Recognition of the need for a standard for writing Distributed memory, parallel programs.
- **April, 1992**: A working group established.
- **November 1992**: MPI draft proposal (MPI1). Group adopts procedures and organization to form the MPI Forum. MPI Forum eventually comprised of about 175 individuals from 40 organizations including.
- **November 1993**: Draft MPI standard presented.
- **May, 1994**: Final version of draft released.
- **1996**: The original MPI (MPI-1) was finalized.
- **Today**: MPI implementations are a combination of MPI-1 and MPI-2.
Reasons for Using MPI

- **Standardization** – MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** – There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** – Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** – Over 115 routines are defined in MPI-1 alone.
- **Availability** – A variety of implementations are available, both vendor and public domain.

General Program Anatomy

- `void MPI::Init(int& argc, char**& argv)`
  - Initializes the execution environment.
  - Must be called in every MPI program.
  - Must be called before other functions.
  - Called only once.

- `int MPI::Comm::Get_size() const`
  - Determines the number of processes in the group associated with a communicator.

- `int MPI::Comm::Get_rank() const`
  - Determines the rank (task ID).
  - A unique integer starting from 0.
  - If a process associated with many communicators, a unique rank within each.

- `void MPI::Finalize()`
  - Terminates the MPI execution.
  - Should be the last MPI called routine.
Communicators are objects to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.

MPI::COMM_WORLD: the predefined communicator that includes all of your MPI processes.
Message = Data + Envelope

- MPI::Comm::Send(const void* buf, int count, MPI::Datatype& datatype, int dest, int tag)
- MPI::Comm::Recv(void* buf, int count, MPI::Datatype& datatype, int source, int tag, MPI::Status* status) const
  or
- MPI::Comm::Recv(void* buf, int count, MPI::Datatype& datatype, int source, int tag) const

Predefined MPI Types

- Data types corresponding to standard C++ types:
  - MPI::INT
  - MPI::SHORT
  - MPI::LONG
  - MPI::LONG_LONG_INT
  - MPI::UNSIGNED
  - MPI::UNSIGNED_LONG
  - MPI::UNSIGNED_SHORT
  - MPI::FLOAT
  - MPI::DOUBLE
  - MPI::LONG_DOUBLE
  - MPI::CHAR
  - MPI::UNSIGNED_CHAR
Wildcards

- **MPI::ANY_SOURCE:**
  - If a process is ready to receive a message from any sending process.
  - Use **MPI::Status** to get information on received information.

- **MPI::ANY_TAG**

---

Example: Parallel Array search

- **Problem:**
  Given an array of numbers, search it for occurrences of a target value. Design a parallel solution using the Message passing Paradigm.
Master Process

- Generate data
- Read Target value

Distribute data & target to workers

Workers

- Search array
- Return result back to master

- Collect results
- Display Output

That’s all for today

Thanks!!