Cloud Computing
CS 15-319
Programming Models- Part I
Lecture 4, Jan 25, 2012

Majd F. Sakr and Mohammad Hammoud
Today…

- Last 3 sessions
  - Administrivia and Introduction to Cloud Computing
  - Introduction to Cloud Computing and Cloud Software Stack
  - Course Project and Amazon AWS

- Today’s session
  - Programming Models – *Part I*

- Announcement:
  - Project update is due today
Objectives

Discussion on Programming Models

Why parallelism?

Parallel computer architectures

Traditional models of parallel programming

Examples of parallel processing

Message Passing Interface (MPI)

MapReduce

Pregel, Dryad, and GraphLab
Amdahl’s Law

- We parallelize our programs in order to run them faster

- How much faster will a parallel program run?

  - Suppose that the sequential execution of a program takes $T_1$ time units and the parallel execution on $p$ processors takes $T_p$ time units

  - Suppose that out of the entire execution of the program, $s$ fraction of it is not parallelizable while $1-s$ fraction is parallelizable

  - Then the speedup (**Amdahl’s formula**):

$$\frac{T_1}{T_p} = \frac{T_1}{T_1 \times s + T_1 \times \frac{1-s}{p}} = \frac{1}{s + \frac{1-s}{p}}$$
Amdahl’s Law: An Example

- SUPPOSE THAT 80% OF YOUR PROGRAM CAN BE PARALLELIZED AND THAT YOU USE 4 PROCESSORS TO RUN YOUR PARALLEL VERSION OF THE PROGRAM

- THE SPEEDUP YOU CAN GET ACCORDING TO AMDALH IS:

\[
\frac{1}{s + \frac{1-s}{p}} = \frac{1}{0.2 + \frac{0.8}{4}} = 2.5 \text{ times}
\]

- ALTHOUGH YOU USE 4 PROCESSORS YOU CANNOT GET A SPEEDUP MORE THAN 2.5 TIMES (OR 40% OF THE SERIAL RUNNING TIME)
Real Vs. Actual Cases

- Amdahl’s argument is too simplified to be applied to real cases.
- When we run a parallel program, there are a communication overhead and a workload imbalance among processes in general.

1. Parallel Speed-up: An Ideal Case

<table>
<thead>
<tr>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>20</td>
</tr>
<tr>
<td>Process 2</td>
<td>20</td>
</tr>
<tr>
<td>Process 3</td>
<td>80</td>
</tr>
<tr>
<td>Process 4</td>
<td>20</td>
</tr>
</tbody>
</table>

2. Parallel Speed-up: An Actual Case

<table>
<thead>
<tr>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>20</td>
</tr>
<tr>
<td>Process 2</td>
<td>20</td>
</tr>
<tr>
<td>Process 3</td>
<td>80</td>
</tr>
<tr>
<td>Process 4</td>
<td>20</td>
</tr>
</tbody>
</table>

© Carnegie Mellon University in Qatar
Guidelines

- In order to efficiently benefit from parallelization, we ought to follow these guidelines:
  
  1. Maximize the fraction of our program that can be parallelized
  2. Balance the workload of parallel processes
  3. Minimize the time spent for communication
Objectives

Discussion on Programming Models

- Why parallelism?
- Parallel computer architectures
- Traditional models of parallel programming
- Examples of parallel processing
- Message Passing Interface (MPI)
- MapReduce
- Pregel, Dryad, and GraphLab
Parallel Computer Architectures

- Multi-Chip Multiprocessors
- Single-Chip Multiprocessors
Multi-Chip Multiprocessors

- We can categorize the architecture of multi-chip multiprocessor computers in terms of two aspects:
  - Whether the memory is physically centralized or distributed
  - Whether or not the address space is shared

<table>
<thead>
<tr>
<th>Memory</th>
<th>Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>SMP (Symmetric Multiprocessor)/UMA (UMA) Architecture</td>
</tr>
<tr>
<td>Distributed</td>
<td>Distributed Shared Memory (DSM)/NUMA (NUMA) Architecture</td>
</tr>
</tbody>
</table>
Symmetric Multiprocessors

- A system with Symmetric Multiprocessors (SMP) architecture uses a shared memory that can be accessed equally from all processors.

- Usually, a single OS controls the SMP system.
Massively Parallel Processors

- A system with a Massively Parallel Processors (MPP) architecture consists of nodes with each having its own processor, memory and I/O subsystem.

- Typically, an independent OS runs at each node.
A Distributed Shared Memory (DSM) system is typically built on a similar hardware model as MPP.

DSM provides a shared address space to applications using a hardware/software directory-based coherence protocol.

The memory latency varies according to whether the memory is accessed directly (a local access) or through the interconnect (a remote access) (hence, NUMA).

As in a SMP system, typically a single OS controls a DSM system.
Parallel Computer Architectures

- Multi-Chip Multiprocessors
- Single-Chip Multiprocessors
Moore’s Law

- As chip manufacturing technology improves, transistors are getting smaller and smaller and it is possible to put more of them on a chip.

- This empirical observation is often called Moore’s Law (# of transistors doubles every 18 to 24 months).

- An obvious question is: “What do we do with all these transistors”?

  Option 1: Add More Cache to the Chip
  - This option is serious
  - However, at some point increasing the cache size may only increase the hit rate from 99% to 99.5%, which does not improve application performance much.

  Option 2: Add More Processors (Cores) to the Chip
  - This option is more serious
  - Reduces complexity and power consumption as well as improves performance.
Chip Multiprocessors

- The outcome is a single-chip multiprocessor referred to as Chip Multiprocessor (CMP)
- CMP is currently considered the architecture of choice
- Cores in a CMP might be coupled either tightly or loosely
  - Cores may or may not share caches
  - Cores may implement a message passing or a shared memory inter-core communication method
- Common CMP interconnects (referred to as Network-on-Chips or NoCs) include bus, ring, 2D mesh, and crossbar
- CMPs could be homogeneous or heterogeneous:
  - Homogeneous CMPs include only identical cores
  - Heterogeneous CMPs have cores which are not identical
Objectives

Discussion on Programming Models

- Why parallelism?
- Parallel computer architectures
- Traditional models of parallel programming
- Examples of parallel processing
- Message Passing Interface (MPI)
- MapReduce
- Pregel, Dryad, and GraphLab

© Carnegie Mellon University in Qatar
Models of Parallel Programming

- What is a parallel programming model?
  - A programming model is an abstraction provided by the hardware to programmers
  - It determines how easily programmers can specify their algorithms into parallel unit of computations (i.e., tasks) that the hardware understands
  - It determines how efficiently parallel tasks can be executed on the hardware

- Main Goal: utilize all the processors of the underlying architecture (e.g., SMP, MPP, CMP) and minimize the elapsed time of your program
Traditional Parallel Programming Models

- Shared Memory
- Message Passing
Shared Memory Model

- In the shared memory programming model, the abstraction is that parallel tasks can access any location of the memory.

- Parallel tasks can communicate through reading and writing common memory locations.

- This is similar to threads from a single process which share a single address space.

- Multi-threaded programs (e.g., OpenMP programs) are the best fit with shared memory programming model.
Shared Memory Model

$S_i = \text{Serial}$

$P_j = \text{Parallel}$

Single Thread

Multi-Thread

Spawn

Join

Shared Address Space
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;

begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
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; // necessary
end parallel // kill the child thread
Print sum;

Sequential

Parallel
Why Locks?

- Unfortunately, threads in a shared memory model need to synchronize.

- This is usually achieved through mutual exclusion.

- Mutual exclusion requires that when there are multiple threads, only one thread is allowed to write to a shared memory location (or the critical section) at any time.

- How to guarantee mutual exclusion in a critical section?
  - Typically, a lock can be implemented.

```c
// In a high level language
void lock (int *lockvar) {
    while (*lockvar == 1) {} ;
    *lockvar = 1;
}

void unlock (int *lockvar) {
    *lockvar = 0;
}
```

```assembly
// In machine language, it looks like this:
lock: ld R1, &lockvar
bnz R1, lock
    st &lockvar, #1
    ret

unlock: st &lockvar, #0
    ret
```

Is this Enough/Correct?
The Synchronization Problem

- Let us check if this works:

  - The execution of `ld`, `bnz`, and `sti` is not atomic (or indivisible)
  - Several threads may be executing them at the same time
  - This allows several threads to enter the critical section simultaneously

```
Thread 0
lock: ld R1, &lockvar
bnz R1, lock
sti &lockvar, #1

Thread 1
lock: ld R1, &lockvar
bnz R1, lock
sti &lockvar, #1

Both will enter the critical section
```
The Peterson’s Algorithm

- To solve this problem, let us consider a software solution referred to as the Peterson’s Algorithm [Tanenbaum, 1992]

```c
int turn;
int interested[n]; // initialized to 0

void lock (int process, int lvar) { // process is 0 or 1
  int other = 1 – process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process, int lvar) {
  interested[process] = FALSE;
}
```
No Race

Thread 0

interested[0] = TRUE;
turn = 0;
while (turn == 0 && interested[1] == TRUE) {};

Since interested[1] is FALSE, Thread 0 enters the critical section

Interested[0] = FALSE;

Thread 1

interested[1] = TRUE;
turn = 1;
while (turn == 1 && interested[0] == TRUE) {};

Since turn is 1 and interested[0] is TRUE, Thread 1 waits in the loop until Thread 0 releases the lock

Now Thread 1 exits the loop and can acquire the lock

No Synchronization Problem

© Carnegie Mellon University in Qatar
With Race

Thread 0

```c
interested[0] = TRUE;
turn = 0;
while (turn == 0 && interested[1] == TRUE) {};
```

Although `interested[1]` is TRUE, `turn` is 1, hence, Thread 0 enters the critical section

- 
- 
- 

```c
interested[0] = FALSE;
```

Thread 1

```c
interested[1] = TRUE;
turn = 1;
while (turn == 1 && interested[0] == TRUE) {};
```

Since `turn` is 1 and `interested[0]` is TRUE, Thread 1 waits in the loop until Thread 0 releases the lock

Now Thread 1 exits the loop and can acquire the lock

No Synchronization Problem

© Carnegie Mellon University in Qatar
Traditional Parallel Programming Models

- Shared Memory
- Message Passing
Message Passing Model

- In message passing, parallel tasks have their own local memories.
- One task cannot access another task’s memory.
- Hence, to communicate data they have to rely on explicit messages sent to each other.
- This is similar to the abstraction of processes which do not share an address space.
- Message Passing Interface (MPI) programs are the best fit with the message passing programming model.
Message Passing Model

\[ S = \text{Serial} \quad P = \text{Parallel} \]

Single Thread

Message Passing

Process 0  Process 1  Process 2  Process 3

Node 1  Node 2  Node 3  Node 4

Data transmission over the Network
Message Passing Example

### Sequential

```c
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;
```

### Parallel

```c
id = getpid();
local_iter = 4;
start_iter = id * local_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);
```

No Mutual Exclusion is Required!
Shared Memory Vs. Message Passing

- Comparison between shared memory and message passing programming models:

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Implicit (via loads/stores)</td>
<td>Explicit Messages</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit (Via Messages)</td>
</tr>
<tr>
<td>Hardware Support</td>
<td>Typically Required</td>
<td>None</td>
</tr>
<tr>
<td>Development Effort</td>
<td>Lower</td>
<td>Higher</td>
</tr>
<tr>
<td>Tuning Effort</td>
<td>Higher</td>
<td>Lower</td>
</tr>
</tbody>
</table>
Objectives

Discussion on Programming Models

- Why parallelism?
- Parallel computer architectures
- Traditional models of parallel programming
- Examples of parallel processing
- Message Passing Interface (MPI)
- MapReduce
- Pregel, Dryad, and GraphLab
- Examples of parallel processing
SPMD and MPMD

- When we run multiple processes with message-passing, there are further categorizations regarding how many different programs are cooperating in parallel execution.

- We distinguish between two models:

  1. Single Program Multiple Data (SPMD) model
  2. Multiple Programs Multiple Data (MPMD) model
In the SPMD model, there is only one program and each process uses the same executable working on different sets of data.
The MPMD model uses different programs for different processes, but the processes collaborate to solve the same problem.

MPMD has two styles, the *master/worker* and the *coupled analysis*.
An Example

A Sequential Program

1. Read array a() from the input file
2. Set is=1 and ie=6 //is = index start and ie = index end
3. Process from a(is) to a(ie)
4. Write array a() to the output file

- Colored shapes indicate the initial values of the elements
- Black shapes indicate the values after they are processed
An Example

**Process 0**
1. Read array a() from the input file
2. Get my rank
3. If rank==0 then
   - is=1, ie=2
   - If rank==1 then
     - is=3, ie=4
     - If rank==2 then
       - is=5, ie=6
4. Process from a(is) to a(ie)
5. Gather the results to process 0
6. If rank==0 then write array a() to the output file

**Process 1**
1. Read array a() from the input file
2. Get my rank
3. If rank==0 then
   - is=1, ie=2
   - If rank==1 then
     - is=3, ie=4
     - If rank==2 then
       - is=5, ie=6
4. Process from a(is) to a(ie)
5. Gather the results to process 0
6. If rank==0 then write array a() to the output file

**Process 2**
1. Read array a() from the input file
2. Get my rank
3. If rank==0 then
   - is=1, ie=2
   - If rank==1 then
     - is=3, ie=4
     - If rank==2 then
       - is=5, ie=6
4. Process from a(is) to a(ie)
5. Gather the results to process 0
6. If rank==0 then write array a() to the output file

© Carnegie Mellon University in Qatar
Concluding Remarks

- To summarize, keep the following 3 points in mind:

  - The purpose of parallelization is to reduce the time spent for computation

  - Ideally, the parallel program is $p$ times faster than the sequential program, where $p$ is the number of processes involved in the parallel execution, *but this is not always achievable*

  - Message-passing is the tool to consolidate what parallelization has separated. It should not be regarded as the parallelization itself
Next Class

Discussion on Programming Models

Why parallelism?
Parallel computer architectures
Traditional models of parallel programming
Examples of parallel processing
Message Passing Interface (MPI)
MapReduce
Pregel, Dryad, and GraphLab