CS3211 Revision - Last lecture

Abhik Roychoudhury
CS 3211
National University of Singapore

Summary of previous 12 lectures

- Concurrency as a concept.
  - Concurrent program execution – inter-leavings
  - Critical section and ensuring mutual exclusion
    - Semaphores, Monitors
  - Deadlocks, Starvation and preventing them.
  - Promela modeling language, process equations, finite state machines.
- Concurrent programming
  - All of the above concepts as evidenced in multi-threaded Java
- Parallel programming
  - Message passing model studied via MPI

Exercise 1

Consider the following MPI program fragment involving two processes. Will there be any deadlocks, or will the two processes progress to completion?

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0){
    MPI_ISend(sbuf, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &req);
    MPI_Recv(rbuf, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1){
    MPI_ISend(sbuf, count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &req);
    MPI_Recv(rbuf, count, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}
MPI_Wait(&req, &status);
```


Answer

- Due to the non-blocking sends, the program avoids deadlocks. Both the processes can post their non-blocking sends and proceed to execute the receive actions. The receives return when the messages have been received.

Exercise 2

- Consider the following MPI program fragment involving two processes. Will there be any deadlocks, or will the two processes progress to completion? Give detailed explanation.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0){
    MPI_Send(a, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    MPI_Send(b, count, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1){
    MPI_Irecv(a, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &req);
    MPI_Recv(b, count, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    MPI_Wait(&req, &status);
}
```


Answer

- The program is safe, that is, it does not deadlock. Process 0 posts two blocking sends. Process 1 posts a non-blocking receive followed by a blocking receive. The first send of process 0 is guaranteed to complete. Thus process 0 can continue and post the second send. Meanwhile process 1 completes its blocking receive and the Wait call ensures that non blocking receive completes as well. Thus, the two processes progress to completion.

Exercise 3

- Consider the following schematic code for the Dining Philosophers' problem.
  - Note that in the schematic code

```
wait_on_cond(Cond){
    append p, the current process to queue for Cond
    p.state = blocked
    monitor.lock = released
}
signal_to_cond(Cond){
    if queue for Cond != empty{
        remove head of queue, let it be process x;  x.state = ready
    }
}
```


Monitor – Dining Philosophers

```
monitor Fork{
   int array[0..4] fork = [2,2,2,2,2]
   condition array[0..4] OKtoEat

   operation takeForks(int i){
     if (fork[i] != 2){
        wait_on_cond(OKtoEat[i])
     }
     fork[i+1] = fork[i+1] - 1;
     fork[i-1] = fork[i-1] – 1;
   }

   operation releaseForks(int i){
      fork[i+1] = fork[i+1]+ 1;
      fork[i-1] = fork[i-1]+ 1;
      if (fork[i+1] == 2){
        signal_on_cond(OKtoEat[i+1])
      }
      if (fork[i-1] == 2){
        signal_on_cond(OKtoEat[i-1])
      }
   }
```

Philosopher i's code
loop forever{ takeForks(i); EAT; releaseForks(i); }

Questions

- Explain the working of the code.
- Does the code suffer from deadlocks?
- Does it suffer from starvation?
- Can you show any of the following
  - eating[i] ⇒ (fork[i] == 2)
    - eating[i] is true when philosopher i has executed takeForks(i), and has not yet executed releaseForks(i).
  - ¬empty(OKtoEat[i]) ⇒ (fork[i] < 2)
  - $\sum_0^4$ fork[i] == 10 – 2 * E,
    - where E == # of phil. who are eating

No deadlock

- Deadlock implies E == 0
- Then fork[0] + fork[1] + fork[2] + fork[3]+fork[4] == 10
- Also, in a deadlock all philosophers should be enqueued on OKtoEat.
- Thus, for all I, fork[i] < 2
  - Hence fork[0] + fork[1] + fork[2] + fork[3]+fork[4] < 10
- Contradiction!

Starvation scenario

phil1	phil2	phil3
take(1)		
		take(3)
	wait(OK[2])	
release(1)		
take(1)		
		release(3)
		take(3)

forever

Exercise 4

```
int x, y, z;   /* MPI_COMM_WORLD = {0,1,2} */
switch (rank) {
   case 0:  x = 0; y = 1; z = 2;
            MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
            MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD); break;
   case 1: x = 3; y = 4; z = 5;
            MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
            break;
   case 2:  x = 6; y = 7; z = 8;
            MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD, &status);
            MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD); break;
}
```

Rank	x	y	z
0	0	1	4
1	0	4	5
2	1	4	0

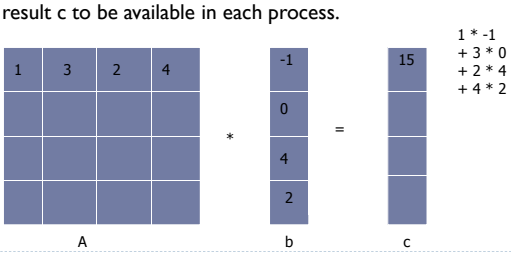
What are the values of x, y, z in the individual processes at the end?

Answer

- Rank x y z
- 0 0 1 4
- 1 0 4 5
- 2 1 4 0
- Explain the reason behind each of the 9 values!

Ex. 5 Matrix-vector mult. in parallel

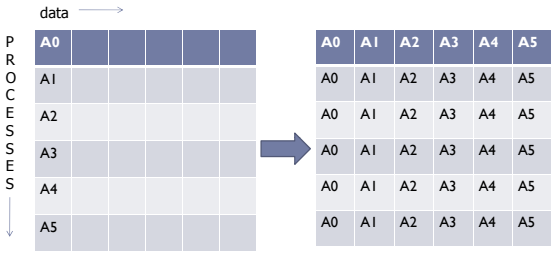
- In class, we discussed dot product computation where two vectors were multiplied. Now, consider the multiplication of a matrix with a vector. We want the result c to be available in each process.



How to divide up the data?

- We are performing A*b = c
  - Assume that rows of the matrix are distributed into proc.
  - Vector b is replicated into all processes.
- Steps
  - Perform local sum (row i of A) * b = element i of c
  - Allgather MPI communication to gather all elements of c.

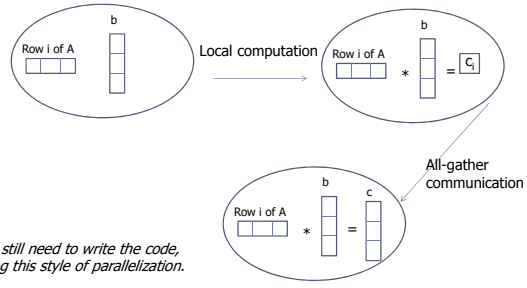
Allgather



Gather to All

- MPI_Allgather(sendbuf, sendcount, sendtype,
- recvbuf, recvcount, recvtype,
- comm)
- There is no root process.
- All-to-all communication.
- All processes receive the gathered result, rather than only the root process.
- As if all the N processes executed N calls to MPI_Gather with root = 0,1,…, N-1.

Pictorially



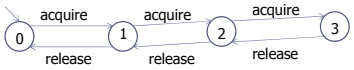
You still need to write the code, using this style of parallelization.

Exercise 6

- The following process equations define a recursive lock (as employed in the Java programming language) which allows a thread to lock a shared object at most k times (where k 2).
  - Klock(x,k) = (when (x < k) acquire -> Klock(x+1,k)
  - | when (x > 0) release -> Klock(x-1,k)
  -).
- Draw the state model for the process Klock(0,3). What do the states of this state machine signify?

Answer

- The states capture the number of acquire actions for which release has not been performed.



Exercise 7

```
synchronized method_in_which_P1_waits(){
       while (!flag1) wait();
       ...
}
synchronized method_in_which_P2_waits(){
          while (!flag2) wait();
          ...
}
synchronized method_which_tries_to_move_ahead(){
    int oracle_says_yes;
    oracle_says_yes = .... // read in integer value
    if (oracle_says_yes > 0) flag1 = true;
    else flag2 = true;
    notifyAll();
}
```

What will happen if we replace notifyAll with notify?

Answer

- The wrong process may be awakened by the scheduler, and it will go back to wait again.
- The process which can get past the wait statement will however remain non schedulable and can suffer from starvation (in terms of being eligible for execution).
  - Lost notification

Exercise 8

In class, we studied how processes can be connected together by relabeling of action names. Consider a process P which has two input actions which input data and acknowledgment, and two output actions which output data and acknowledgment.

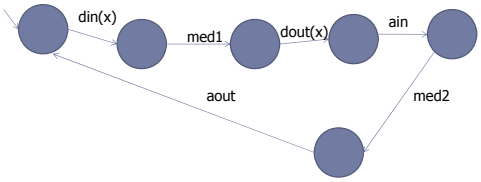
- P = din(x) -> dout(x) -> ain -> aout -> P

How can you connect two such processes, such that (i) the dataout of the first process is fed to the datain of the second process, and (ii) the ackout of the second process is fed to the ackin of the first process? Note that the datain/ackin of the first process as well as the dataout/ackout of the second process should be externally visible.

Write the process equation for the resultant process.

Compare the resultant process with P in terms of allowed execution traces.

Answer



(P/{med1/dataout, med2/ackin} || P/{med1/datain, med2/ackout})\{med1,med2}

The resultant process is equivalent to P, as can be confirmed by drawing the state models of P and the resultant process. Above is the state model without the action hiding. The action hiding will make it equivalent to P.