End Sem Exam 2020

Name ⇒ Himanshu Gupta
Roll No ⇒ BT17CSE093
Sem, Year ⇒ 7th Sem
Branch ⇒ CSE
Course ⇒ IPC
Date ⇒ 26/11/2020

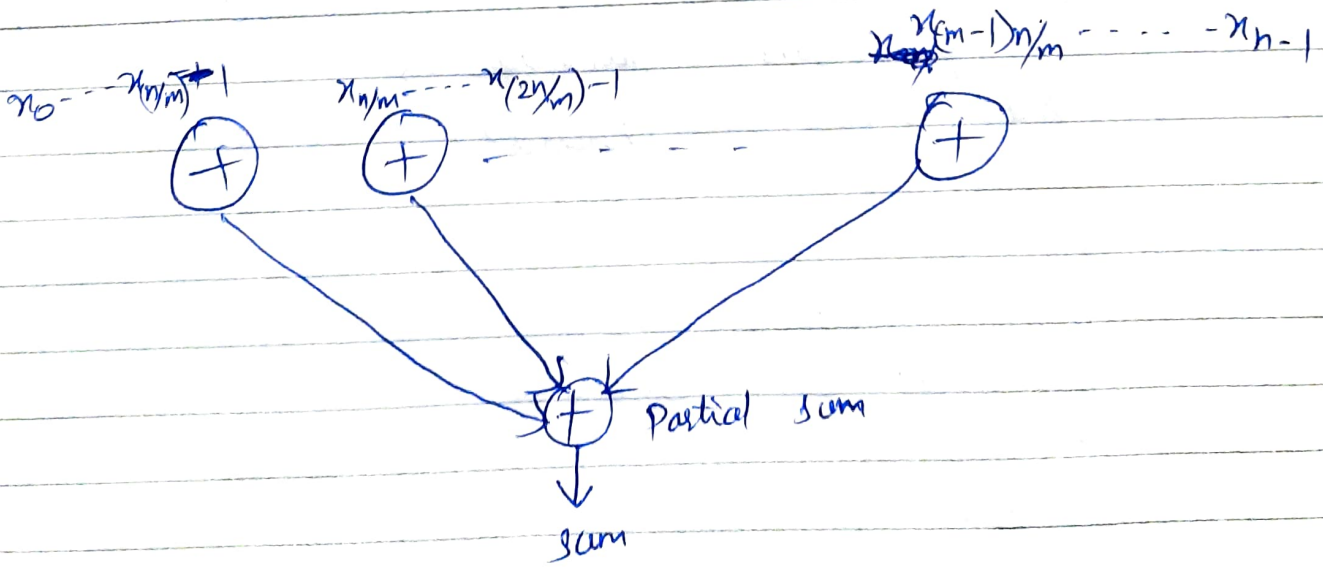
Q2) Using Partitioning strategy to solve the addition of numbers $x_0, x_1, \ldots, x_{n-1}$

a) Steps -

(1) Divide sequence into m parts of n/m numbers each and distribut ~~pro~~ sequences to corresponding processors.

(2) m processors can ~~add~~ each add one sequence independently to create partial sums

(3) partial sums are added together on master to form final sum.



Sequential code

The code that uses seperate send() and recv()s.

Master:

```
s = n/m;                    /* number of items on each slaves */
for(i=0, x=0; i<m; i++, x=x+s)
    send(&numbers[x], s, P_i);

sum=0;
for(i=0; i< m; i++) {       /* wait for results from slaves */
    recv(&part_sum, P_ANY);
    sum += part_sum         /* accumulate partial sums */
}
```

Slave:

```
recv(number, s, P_master);  // receive s items from master
part_sum = 0;
for(i=0; i<s; i++)          /* add numbers */
    part_sum += numbers[i];
send(&part_sum, P_master);  /* send result to master */
```

Using Broadcast:

The code using Broadcast routine:-

Master:

```
s = n/m;                    // no. of numbers for slaves
bcast(numbers, s, slave_group);
sum = 0;
```

```
for(i=0; i<m; i++) {        /* wait for result from slaves */
    recv(&part_sum, P_ANY);
    sum += part_sum;
}
```

Slave:-

```
bcast(number, s, P_master);   /* receive all numbers from master */
start = slave_number * s;     // slave no. obtained earlier
end = start + s;
part_sum = 0;
for(i = start; i<end; i++)
    part_sum = part_sum + numbers[i];   // add numbers
send(&part_sum, P_master);    // send sum to master.
```

So, in this sequential & parallel implementation is done using master & slave approach

$$\text{Speedup} = \frac{t_s}{t_p} = \frac{n-1}{n/m + m - 2}$$

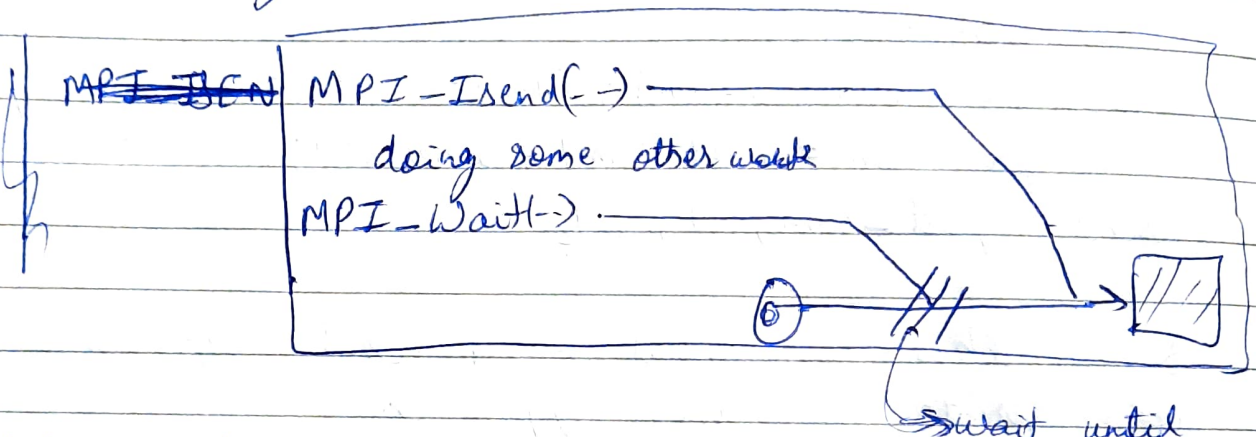
Ques5) Blocking & Non-Blocking communication in MPI

Blocking communication is done using MPI_Send() and MPI_recv(). These functions do not return (i.e., they block), until the communication is finished. Similifying somewhat, this means that the buffer passed to MPI_SEND() can be reused, either because MPI saved it somewhere or because it has been received by the destination. Similarly, MPI_RECV() returns when buffer has been filled with valid data.

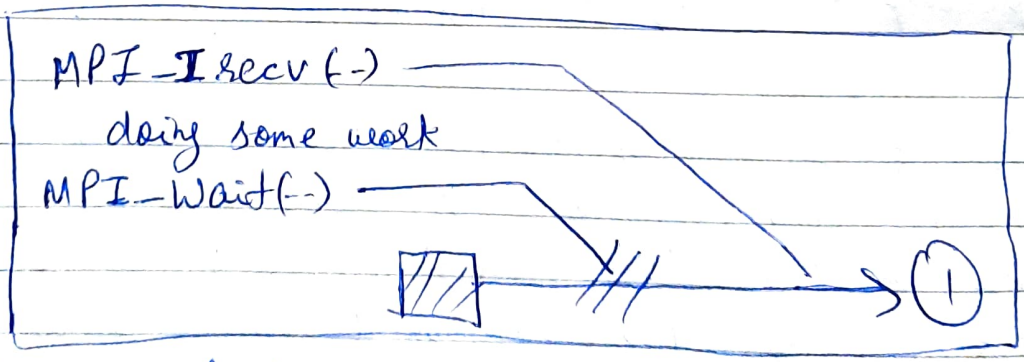
In contrast, non-blocking communication is done using MPI_Isend() and MPI_IRECV(). These function return immediate (i.e. they don't block) even if communication is not finished yet. you must call MPI_WAIT() or MPI_TEST() to see whether the communication has finished.

Example

(1) Non-Blocking Send :



(2) Non-Blocking Receive :



In C :- other MPI functions

(i) MPI_Isend (buf, count, datatype, dest, tag, comm, OUT &request_handle);
MPI_Wait(INOUT &request_handle, &status);

(ii) MPI_Irecv(--)

(iii) MPI_Ibsend(---) → ~~Synch~~ buffered.

(iv) MPI_IRsend(--)

Teacher's Signature

(v) MPI_MAX()
(vi) MPI_MIN()
(vii) MPI_SUM()
(viii) MPI_PROD()
(ix) MPI_LAND()
(x) MPI_BAND()

6) Assume size of A, B and C is 1000 elements.

(a) Thread blocks will generate = 4

(b) No. of wraps in each block = 8

(c) Yes, there is control divergence, which is caused by the if statement in line 2.

```
if (i<n) c_d[i] = A_d[i] + B_d[i];
```

Since the total number of threads in the grid will be 1024, larger than the size of arrays, the last warp will have divergence. First 8 thread in wrap will take the true path and ther other 24 threads will take false path.

7)

```
for (i=0; i<=100; i++)
    for(j=4; j<=70; j++)
        H[i,j];
```

(i) S: A[i+2, j] = B[2i, j] - 3;
T: B[2j, j-1] = A[i, j+2] + 12;

for S = (0, 0)

------ = B[6, 4]

B[6, 3] = ------

So, Here is anti-dependence of S on T

i.e. $S \delta^a T$

distance vector = (0, 1)
direction vector = (0, +)
level = 2.

for S = (3, 6)

A[5, 6] = ...

= A[3, 8]

Hence, it is flow dependent

i.e. $S \delta^f T$

Now,

distance vector = (2, -2)
direction vector = (1, -1)
level = 1

(ii) ~~S: A[i+2, j] =~~

(ii) S': A[i, j] = B[i+4, j-2] + B[i+2, j-3] + B[i, j+3];

T: B[i, j] = C[i, j] + A[i+1, j-2] + 2;

~~for~~

for S(3, 4)

S: ... = B[3, 7]

T: B[3, 4] = ...

So,

Here is S δ^a T (anti-dependence)

distance vector = (0, 3)

direction vector = (0, 1)

level = 2.

T(3, 5) δ^a S(4, 4) for S: A[4, 4] = ...

.... A[5, 3]

distance vector = (1, -1)

direction = (1, -1)

level = 1

S(3, 6) δ^a T(7, 4)

S: .... = B[7, 4]

T: B[3, 6]

distance vector = (4, -2)

direction = (1, -1)

level = 1

S(3, 7) δ T(5, 4)

S: .. = B[i+2]

T: B[i, j] = ...

distance vector = (2, -3)

direction = (1, -1)

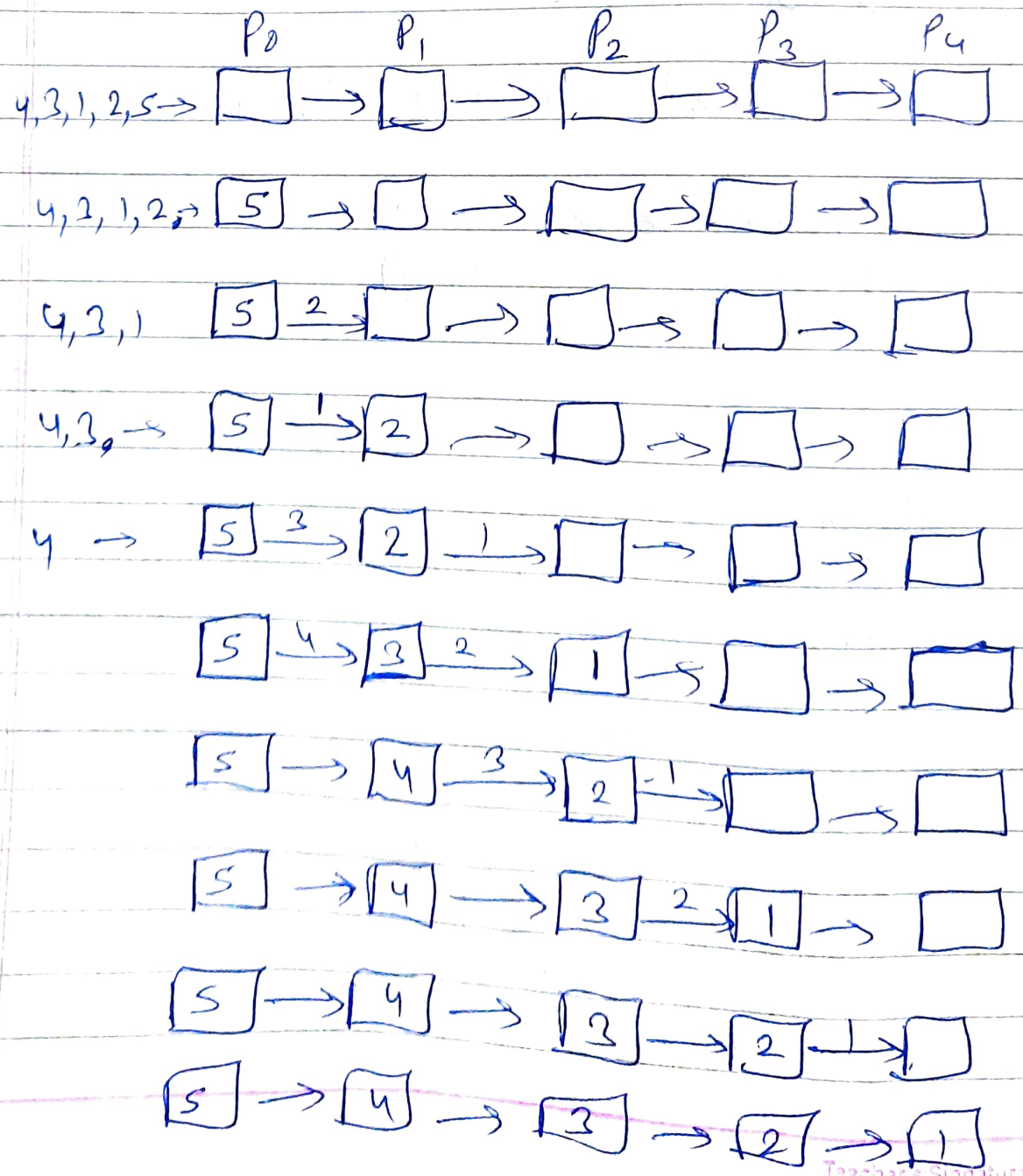
level = 1

Ans 4) We need to sort the elements of array. We can use Insertion sort with bits line.

Algorithm:-

1. Insert data elements to one of the processor
2. Each processor on sorting element will compare itself with given number.
3. If given number is large, it will keep the number and pass smaller number
4. else, pass the given number.

Example: array = [4, 3, 1, 2, 5] & no. of processors = 5



Assuming compare & exchange as single computational step

$$t_s = (n-1) + (n-2) + \cdots + (2) + 1 = \frac{n(n-1)}{2}$$

whereas, parallel Algorithm has

$$n + n - 1 = 2n - 1 \quad \text{(pipeline cycles)}$$

$$t_{computation} = 1$$

$$t_{communication} = 2(t_{comp} + t_{data})$$

&

&, $t_p = O(n)$

and serial, $T_s = n^2$

$$\text{Speedup} = \frac{T_s}{T_p} = \frac{n^2}{n} = n$$

No. of Assignment = 4950

② schedule (static, 1)

thread 0 : [0] [4] [8]

thread 1 : [1] [5] [9]

thread 2 : [2] [6] [10]

thread 3 : [3] [7] [11]

issue:- Not utilising the cache well. 0,1,2,3 will be stored nearby (assuming row-major form in storing numbers)

(1) ~~Schedule~~ (Static, 10)

thread 0: 0-9 | 40-49 | 80-89

thread 1: 10-19 | 50-59 | 90-99

thread 2: 20-29 | 60-69 | 100-109

thread 3: 30-39 | 70-79 | 110-119

issue: Does not take care of the dynamic nature (i.e. which computation takes more time, which takes less, allocating sets accordingly).

(3) schedule(dynamic, 10)

thread 0: 0-9 | 40-49 | 60-69

thread 1: 10-19 | 50-59 | 70-79 | 100-109

thread 2: 20-~~29~~ 29 ~~60-69~~ | 90-99

thread 3: 30-39 | ~~60~~-69 | 80-89

→ More balanced allotment.

(4) Schedule(dynamic, 20)

thread 0 :- 0-19

1 : 20-39

2 : 40-59

3

3 : 60-79

Note :-

Chunk size ↑
Static size ↑
dynamic nature ↓