Date _____ Day _____

"PARALLEL SOFTWARE"

CAVEATS:

Single Program Multiple Data:

↳ Same program working on different parts of data.

data ||ism

```
if (I am thread/process 0)
    Operate on first half of array;
else
    Operate on second half
```

task ||ism

```
if (I am thread/process 0)
    do this;
else
    do this;
```

→ Divide the work among process/threads in such a way that each process/thread gets roughly same amount of work, is called load balancing.

→ Programs that can be ||ised by simply dividing the work among process/threads are called embarrassingly parallel.

Date _____ Day _____

SHARED MEMORY :

→ Communication is done thru shared & private variable

→ Dynamic & Static Threads :

↳ In Dynamic threads program, there is a master thread and at any given instant a (possible empty) collection of worker threads.
When a request arrive for a work to be done master forks the worker thread, which after completion of task joins the master thread.

↳ In Static Thread paradigm, all of the threads are forked and all the threads run untill all the work is completed. After completion of work they join master thread and then they are freed.
If a thread is idle, it resource can't be freed.

→ Non determinism :

↳ A computation is non-determinism if an input can result in diff outputs.

printf("Thread %d, my_val = %d \n", my_rank, my_

The order in which threads execute is diff in each run.

But in shared memory system this problem can lead to race condition

Date _____ Day _____

↳ To avoid threads to simultaneously access critical section we use mutual exclusion lock or mutex or lock.

↳ The use of mutex enforces serialization of critical section.

↳ Busy waiting is an alternative to the mutex.
Wastes system resources / cpu cycles

```
my_val = Compute_val (my_rank);
if (my_rank = 1)
    while (! ok_for_1);

n += my_val;
if (my_rank == 0)
    ok_for_1 = true;
```

↳ Semaphores are similar to mutexes

↳ A monitor provides mutual exclusion at higher level. It is an object whose method can only be executed by one thread at a time.

↳ Transactional Memory : In DBMS, a transaction is an access to database that the system treats as a single unit. It either completes sucessfully or if it encounters any error during transaction it rolls back erasing all the data.

Date Day

→ Thread Safety :

Example : The C library function strtok splits an input string into subsequent strings. When it's first called it's passed a string, and on subsequ calls it returns successive substring.

If thread 0 makes it 1st call to strtok, and then thread 1 makes it's 1st call to it, before t0 has completed splitting it's string, then t0's string will be lost or overwritten, and on subsequ calls it may get substring of t1's strings.

So strtok is not thread safe.

▷ DISTRIBUTED - MEMORY :

→ Message Passing :

```
char message[100];

my_rank = Get_rank();
if (my_rank == 1){
    sprintf(message, "Greetings from process 1");
    Send(message, MSG_CHAR, 100, 0);
                  Type of element  no. of elements  rank of recieving process
else if (rank == 0){
    Recieve(message, MSG_CHAR, 100, 1);
    printf("Process 0 > Recieved %s\n", message);
}
```

Page No.

Date Day

▷ INPUT AND OUTPUT :

→ In DMS programs, only process 0 will access stdin. In SMS programs, only master thread or thread 0 will access stdin

→ In both DMS & SMS programs, all the process/threads can access stdout & stderr.

→ Only a single process/thread will attempt to access any single file other than stdin, stdout or stderr. For eg: each process can open it's own private file for reading or writing, but no two process/threads will open same file

→ Debug output should always include rank or id of process/thread.

▷ PERFORMANCE - Speedups & Efficiency :

$$\text{Speedup} = \frac{T_{serial}}{T_{parallel}} = S$$

Let p = no. of cores

Then, $T_{parallel} = \frac{T_{serial}}{p}$

Then we say our H program has linear speedup.

Page No.

Date _____ Day _____

Ideally for linear speedup,

$$S = \frac{T_{serial}}{\frac{T_{serial}}{P}} = P$$

The value $\frac{S}{P}$ is called efficiency

$$E = \frac{S}{P} = \frac{T_{serial}}{P \cdot T_{parallel}}$$

S/p will get smaller & smaller as P increases

If we include the necessary "parallel overhead" cause due to parallelizing, because of communication and mutual exclusion then,

$$T_{parallel} = T_{serial} / P + T_{overhead}$$

"AMDAHL'S LAW"

If we are able to ~~U~~ize 90% of a serial program.
If T_{serial} = 20 sec, Then,

$$T_{parallel} = 0.9 \times \frac{T_{serial}}{P} = \frac{18}{P}$$

& For unparallel part, $0.1 \times T_{serial} = 2$

$$T_{parallel} = \frac{18}{P} + 2$$

Page No. ☐

Formula: $S(N) = \frac{1}{\alpha + \frac{(1-\alpha)}{N}}$

where,
α = serial fraction
N = no. of processor

Date _____ Day _____

$$\text{Speedup} = S = \frac{T_{serial}}{\frac{18}{P} + 2} = \frac{20}{\frac{18}{P} + 2}$$

Now if 'P' gets larger than the value 18/p comes closer to zero. This means ~~how~~ regardless of how much we incres. p (no. of cores) the speedup will remain less than 20/2 i.e 10.
Even if we use 1000 cores we cant get speedup better than 10.
It means if we have unparallelized part r, then we can't get speedup better than 1/r. Here r = (1-0.9) = 1/10

GUSTAFSON'S LAW :

This law considers the workload's scalability and disregards the idea of having a fixed portion of the workload.
As the computing resources (typically the no. of processors) increase, the problem size & workload can be scaled up to utilize the available resources effectively.

→ Adv & disadv from PDF. Week#05 & 06

Formula: → $S = 1 + (N-1) \times P$

$S = N + (1-N) \times S$

As we incr the no. of processor (N) the workload or problem size also increases.

N = No. of processor
S = fraction of time for serial part

Page No. ☐

Infinix NOTE

Date _____ Day _____

→ Strongly Scalable : When we incr the no. of process we can keep efficiency fixed without incr the problem size, the program is said to be strongly scalable.

→ If we can keep the efficiency fixed by incr the problemsize at the same rate as we incr the no. of process, then the program is said to be weakly scalable.

Date _____ Day _____

CHAPTER # 03

"Message Passing Interface"

→ Command to run MPI programs

mpiexec -n <no of process> ./mpi_hello
mpiexec -n (4) ./mpi_hello
↳ this program will be executed by 4 threads

▷ MPI_Init & MPI_Finalize :
MPI_Init initializes the setup of message passing like allocating storage for message buffers and might decide which process gets which ranks.

MPI_Init (&argc, &argv)
↳ optional parameters

MPI_Finalize();
↳ tells that we are done with MPI

▷ Communicator, MPI_Comm_size And MPI_Comm_rank :
MPI communicator is a collection of process that can send messages to each other. This communicator is called MPI_COMM_WORLD. Initialize by MPI_Init():

int MPI_Comm_Size (MPI_COMM_WORL, no_of_process);
int MPI_Comm_Rank (, curr_process_rank);

Date Day

→ We use the approach single program multiple data (SPMD), in which we use same program and create diff branches acc to rank.

▷ MPI_Send():

MPI_Send(msg_buffer, msg_size, msg_type, dest, tag, communicator);

tag → It is a non-negative int. It is used to distinguish messages. Suppose P1 is sending floats to P0. Some of floats should be printed while others should be used in computation. So process 1 can use tag of 0 for msgs that should be printed & tag 1 for computational msgs.

Two process having diff communicator can not send & rec msgs.

▷ MPI_Recv():

if recieved msg is larger than buffer then an overflow err will occur & return MPI_ERR_TRUNCATE.

MPI_Recv(msg_buffer, msg_siz, ~~msg~~ type, source, tag, communicator, status_p);

This can happen some times that we don't know in which order the msgs will be recieved. So instead

Page No.

Date Day

of writing source we can use MPI_ANY_SOURCE same for tag MPI_ANY_TAG.
↳ But these are not valid for MPI_Send();

▷ status_p Argument:

→ MPI_Status is a struct with at least three members MPI_SOURCE, MPI_TAG and MPI_ERROR.
→ We have to make a variable MPI_Status status.
→ This is passed as &status last argument of MPI_Recv.
→ After that we can access the source thru status.MPI_SOURCE and tag thru status.MPI_TAG.
→ To get the amount of data recieved we have to pass this status variable in
MPI_Get_count(&status, rec_type_, &count);
↳ Now the count variable has the amount of data recieved.

Page No.

Infinix NOTE

Date _____ Day _____

▷ Structure of Message-Passing Programming :

① Asynchronous Paradigm :
All concurrent tasks execute asynchronously. This mak
it possible to implement any parallel algo.
These types of programs can have non-determinis
behaviour due to race condition.

② Loosely Synchronous :
In such programs, tasks or subset of tasks
synchronize to perform interactions.

▷ Direct Memory Access :
DMA allows copying of data from one memory locati
to another (eg: communication buffers) without CPU
support.

▷ Blocking Non-Buffered Send & Recieve :

↳ Blocking Non-Buffered Send :
- The send operation is considered blocking when the sending process waits until the message has been fully transmitted to the reciever or until the reciever is ready to recieve the message.
- Non-buffered means the sys doesnot use intermedia

Page No.

Date _____ Day _____

memory (buffer) to temporarily store the message.
- The send oper only completes when the reciever has started to the recieve the data, ensuring that the mesg is imediatly consumed.

↳ Blocking Non-buffered Recieve :
- The recieve operation is considered blocking when the recieving process waits until the msg arrives.
- It will only proceed once the sender has transmitted the data & recievers has successfully obtained it.
- Non-buffered means the system does not allocate a temporary buffer; the reciever waits for the data to be sent directly.

➡ Both the sending & recieving process must be ready because if one isn't ready the other would be to idle (waiting for the other process).

↳ Deadlocks in Blocking Non-Buffered Operations

P_0	P_1
send (&a, 1, 1);	send (&a, 1, 0);
recieve (&b, 1, 1);	recieve (&b, 1, 0);

Done ek dosre ko data send krrhe hai or done wait krrhe hai k data recieve ho dosre process ki taraf se.

Page No.

22K-4187 Samooze Ali

ASSIGNMENT #02

Date Day

This results in indefinite waiting / which results in deadlock.

QUESTION#01 (a)

▷ Blocking Buffered Send / Recieve :

↳ Blocking Buffered Send :

- The sending process uses a buffer to temporarily store the message, allowing the sender to proceed with waiting for the recievers to be ready to accept the msg.
- Once the msg is copied to buffer, the sender can continue with it's execution.
- The process is only blocking until the data is copied to buffer.

↳ Blocking Buffered Recieve :

- The recievers waits for the message to be available, blocking until the msg arrives.
- The msg is transferred from the buffer to recievers memory when the recievers calls the recieve operation.

```
MPI_Buffer_attach (buffer, BUFFER_SIZE);
if (rank == 0) {
    data = 100;
    MPI_BSend (&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf ("Process 0 sent data %d ", data);
}
```

Page No.

Date Day

```
else if (rank == 1) {
    MPI_Recv (&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
    printf ("Process 1 recieved data %d \n", data);
}
MPI_Buffer_detach (buffer, & BUFFER_SIZE);

MPI_Finalize ();
}
```

→ Buffer protocols alleviate idling overheads at the cost of adding buffer management.

→ If program is highly sync, then non-buffered may perform better.

↳ Impact of Finite buffers in Message Passing :

P_0	P_1
for (i=0; i<1000; i++) {	for (i=0; i<1000; i++)
produce_data (&a);	recieve (&a, 1, 0)
Send (&a, 1, 1);	consume_data (&a);
}	

If P_1 was slow in getting loop, the P_0 might have sent all of it's data - If there is enough buffer space

Page No.

Infinix NOTE

Date _____ Day _____

space, then both process can proceed; however if the buffer is not sufficient (i.e buffer overflow), the sender would have to be blocked until some of the corresponding recieve operations had been posted, thus freeing up buffer space.

↳ Deadlock in Buffered Operation:

P_0	P_1
sec(&a, 1, 1)	sed &a, 1, 0);
send(&b, 1, 1)	send (&b, 1, 0);

QUESTION# 2 (a)

▷ Non-Blocking Message Passing:

↳ Non-blocking Send: (Non-buffered)

- The sending process initiates the transmission of a msg & then immediately proceeds with it's computation. The send oper doesnot wait for the recieven to be ready.
- The data might not have been completely transferred when the send operation returns, so the user must ensure that the data buffer is not modified until the communication completes.

Page No.

Date _____ Day _____

↳ Non-blocking recieve:

- The recieving process initiates a request to recieve data but continues its exec without waiting for data to arrive.
- The process can check whether the msg has been recieved at a later point or can proceed with other work.

```
MPI_Request request;

if(rank==0) {
    data = 100;
    MPI_Isend(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
              &request);
    printf("Process 0 initiated send\n");

    // Do some other work
    // Ensure send oper completes
    MPI_Wait(&request, &status); // P0 Completed
}
else if(rank==1) {
    MPI_Irecv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &request); // P1 initiated
    // Do some other work
    MPI_Wait(&request, &status); // P1 completed
}
```

Page No.

Infinix NOTE

Date Day

↳ Buffered :
The msg is temp stored in buffer, allowing the sending process to continue immediately.

" MPI - LIBRARY "

// To get name of the processor.
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;

MPI_Get_Processor_name(processor_name, &nam_len);

// MPI_Send Recv Simultaneously.
int a[10], b[10], npes, myrank;
MPI_Status status;

MPI_Comm_Size(MPI_COMM_WORLD, &npes);
MPI_Comm_Rank(, &myrank);

MPI_Send Recv(a, 10, MPI_INT, (myrank+1)%npes, 1,
b, 10, MPI_INT, (myrank-1+npes)%npes,
MPI_COMM_WORLD, &status);

↳ avoids circular deadlock problem

Date Day

// MPI Send & Recieve ~~for~~ with Single buffer

MPI_Sendrecv_replace (void *buff, int count,
MPI_Datatype datatype, int dest,
Use single buffer for both int send_tag, int source,
send & recieve operations int recv_tag, MPI_COMM_WORLD,
& status);

Recieved data replaces the data that was sent out of the buffer.

// Checking whether a non-blocking oper completed?
MPI_Test(MPI_Request *req, int *flag, & status);
It tests whether or not the operation send/recv identified by its req has finished. And returns true in flag otherwise false.

// non-blocking operation has finished, the request object is deallocated & req is set to MPI_~~Request~~_NULL
REQUEST

// For explicitly deallocating req object.
MPI_Request_free (&request);

Date _____ Day _____

- MPI_Barrier() enforces sync among all process in a communication world by making them wait until every processs reaches the barrier, ensuring coordination at specific program points.

MPI_Barrier(MPI_COMM_WORLD);

- MPI_Bcast() → A broadcast func is one of the std collective communication technique. One process sends the same data to all process in a communicator.
The process designated as "root" sends the msg.

MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD)
is data variable mein jo bhi data hoga we sabko milega sender of process that sends the data

↳ Useful when one process has data that needs to be distributed (shared with all other processes), such as user input parameter or configuration parameters

- MPI_Scatter() → Similar to broadcast but used to distribute different chunks of data from root process to all other process in a communicator, including itself

Page No.

Date _____ Day _____

MPI_Scatter(void *sendbuffer, int sendCount, MPI_Datatype send datatype, void *recbuffer, int recCount, MPI_Datatype recvtype, int root, MPI_COMM)

- If we have an array = {10, 20, 30, 40, 50, 60} then root P_0 will recieve 10, P_1 recieve 20, P_2 30, P_3 40 & so on. (Pos sendCount = 1)
- If sendCount = 2 each process recieves 2 elements.
- Usually sendCount = no of element / no of process.

- MPI_Gather() → It is inverse of scatter. It takes elements from many process and gathers them to one single process.
It takes elements from each process & gathers them to the root process.
Same Syntax as scatter.
↳ Here only root process needs to have a valid ~~buffer~~ recieve buffer, other calling process can pass NULL for rec_data.
↳ Here recCount is the count of elements recieved per process, not the total summation of counts from all process.

Page No.

Day

Date

- MPI_Allgather() → like for many to many communication
Given set of elements distributed across all process MPI_Allgather will gather all of the elements to all the processes.

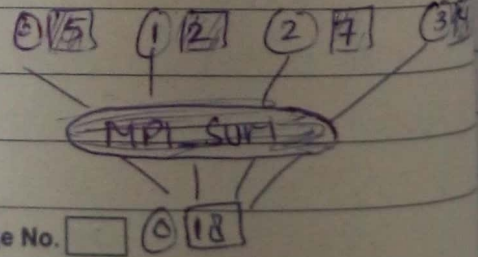
```
MPI_Allgather(void *sendbuf, int sendCount,
              MPI_Datatype sendType, void *recbuf,
              int recCount, MPI_Datatype recType,
              MPI_COMM_WORLD);
```

- MPI_Reduce() → Works similar to gather, the only difference is that it returns the result to the root process by doing some operation on data.

```
MPI_Reduce(void *send_data, void* rec_data, int count,
           MPI_Datatype datatype, MPI_Op op,
           int root, MPI_COMM_WORLD);
```

Some predefined operation can be applied.

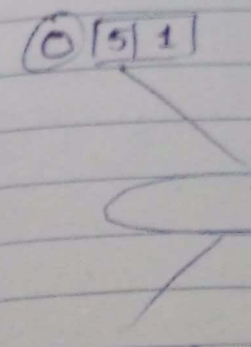
MPI_MAX	MPI_BXOR
MPI_MIN	MPI_MAXLOC (return max val & rank of process that owns it)
MPI_SUM	MPI_MINLOC
MPI_PROD	
MPI_LAND (Logical AND)	
MPI_LOR	
MPI_BAND (Bitwise AND)	
MPI_BOR	
MPI_LXOR	



Page No.

Date

- MPI_Allreduce :

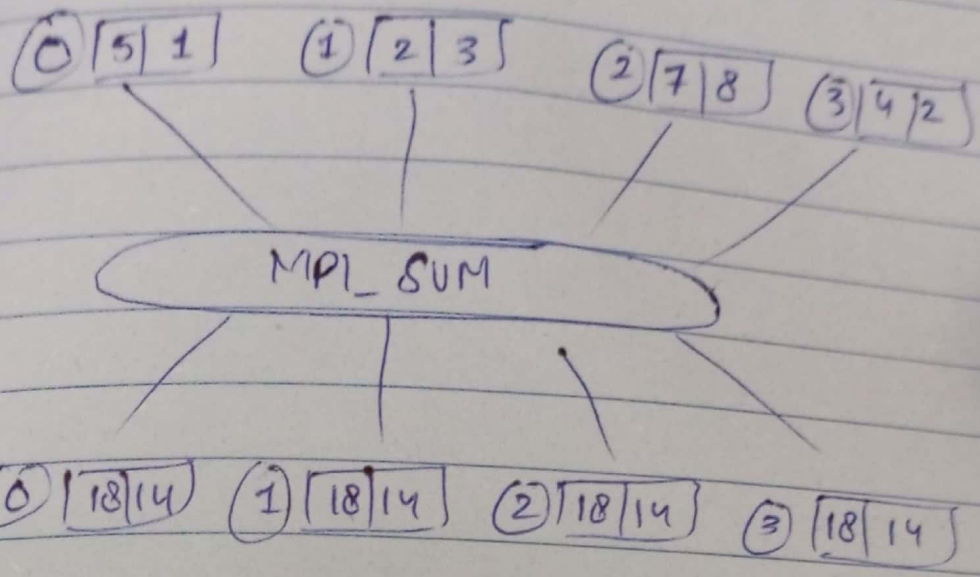


MPI_Allreduce

- MPI Time
Use to ca
MPI Progra
double d1
d1 = MPI_
d2 = MPI_
printf("Ti

Date Day

• MPI_Allreduce :




```
MPI_Allreduce(void *sendbuf, void *recbuf, int count,
              MPI_Datatype datatype, MPI_Op op,
              MPI_COMM_WORLD);
```

• MPI Timer :

Use to calculate elapsed time b/w 2 points in an MPI Program.

```
double d1, d2;
d1 = MPI_Wtime();
:
d2 = MPI_Wtime();
printf("Time : %.f", d2-d1);
```