Date :
Page No.

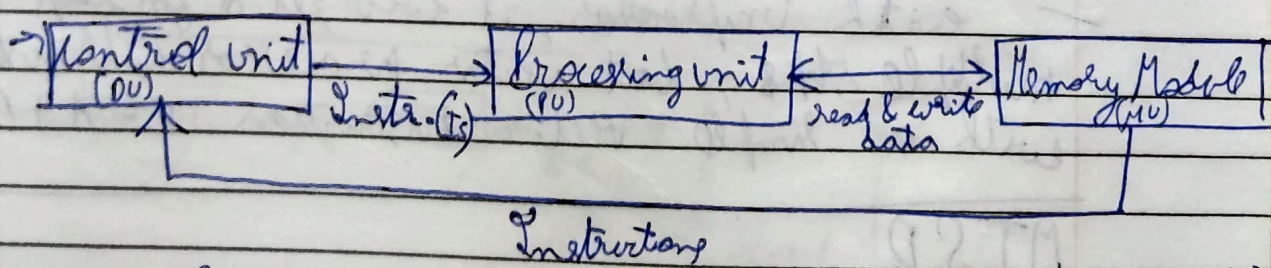
HIGH PERFORMANCE COMPUTING

Flynn's Classification of Parallel Architecture

SISD → Single Inst. & Single data stream
SIMD → Single Inst. & Multiple " "
MISD → Multiple " " Single " "
MIMD → Multiple " " Multiple " "

→ Based on number of instructions & number of data sets

There are processing and data elements. These processing elements are connected.



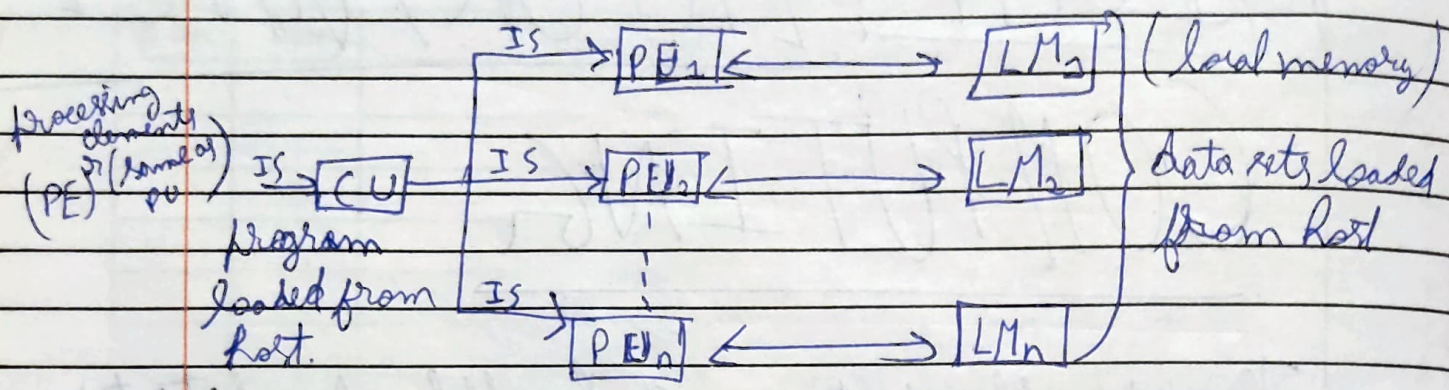
SISD uniprocessor system (e.g. → von neumann computers)

→ upto 8086 were all uniprocessor system.

→ quad core, hexa core → multiprocessor system
↓ 8 processors ↓ 6 processors

Date :
Page No.

SIMD :-



→ every processor may have its own memory module and also cache module

→ same inst. sent to all PEs.

→ kind of master slave architecture.

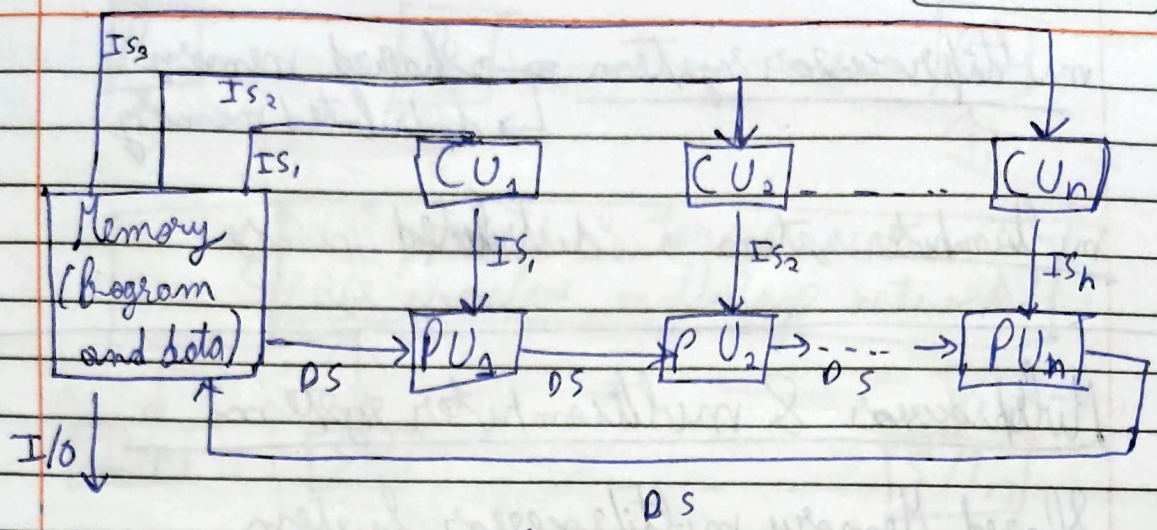
Vector processors → best e.g. of SIMD.
→ They don't work on pair of single elements

e.g.→ if we want to add A[10] & B[10] then with uniprocessor, it will get a for loop & require multiple steps, but vector processor will just do it with a simple vector instruction C = A + B

MISD

→ designed for some specific applications.
→ not quite popular.
→ picks up instructions which are not executed & hence not much useful, hypothetical

Date :
Page No.

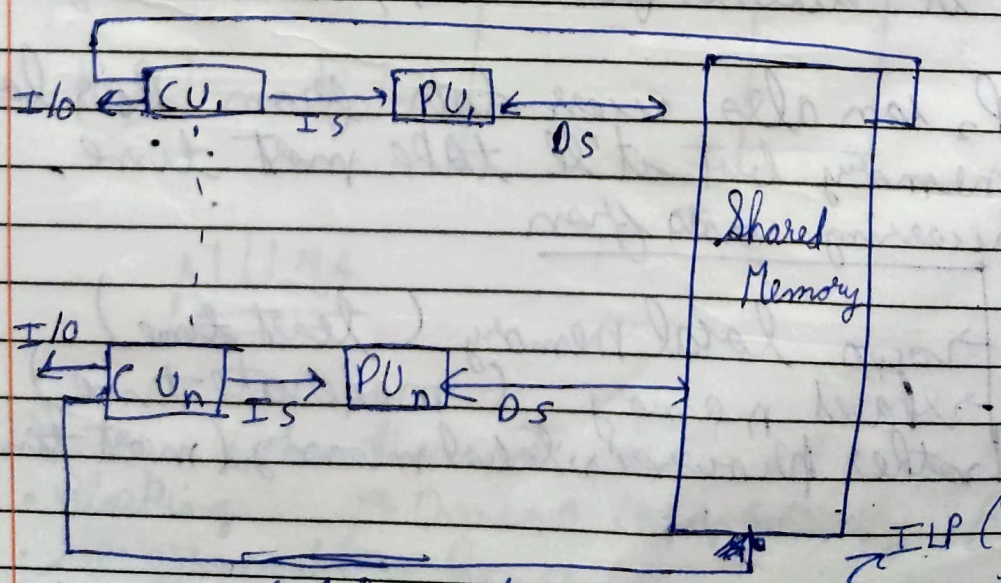


e.g. → Systolic array (Application :- Matrix multiplication)

MIMD (more popular nowadays)

- Multiprocessor system variant
- Multicomputer system variant (more powerful)
  - ↳ individual computers connected through network.

- supercomputers use multiprocessor system.



→ having multiple pipelines → Superscalar computers. ← ILP (Instruction level parallelism)

Date :
Page No.

multiprocessor system → shared memory
→ distributed memory

multicomputer system → distributed memory.

Multiprocessor & multicomputer systems

Shared Memory multiprocessor System

- UMA → uniform memory access
- NUMA → non-uniform memory access
- COMA → cache only memory system

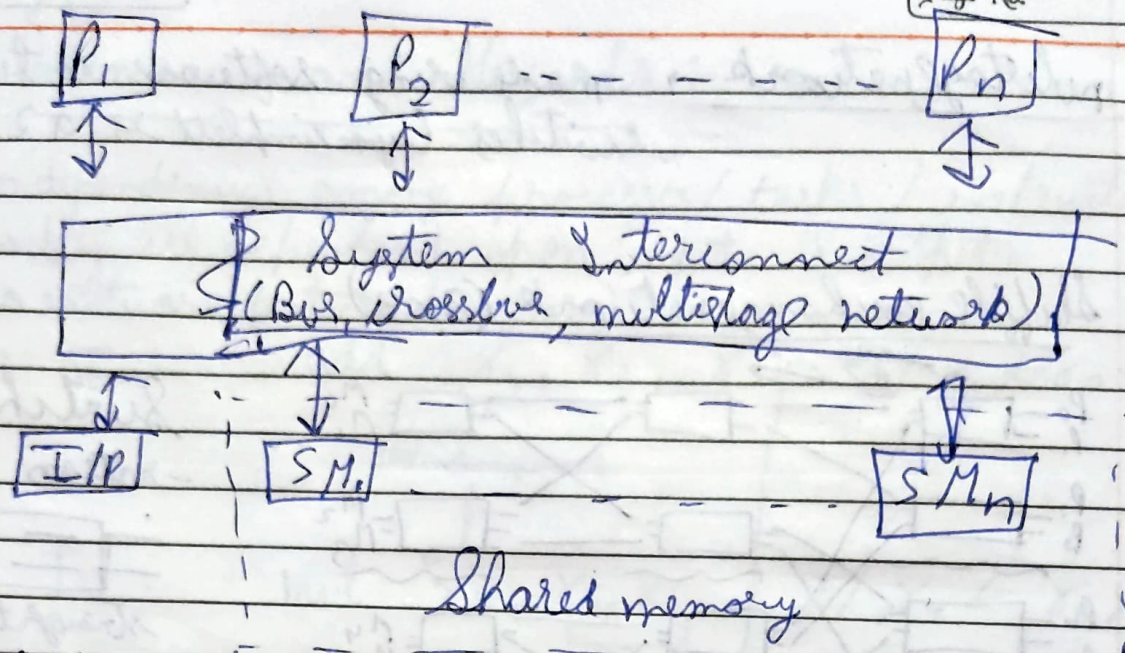
→ UMA → same time taken by any processor to access any data location.

→ NUMA → shared as well as local memory.
→ If processor 1 takes data from shared memory it will take time but much less time if it accesses from its local memory.

→ P_2 can also access data from P_1's local memory but it'll take most time.
accessing data from

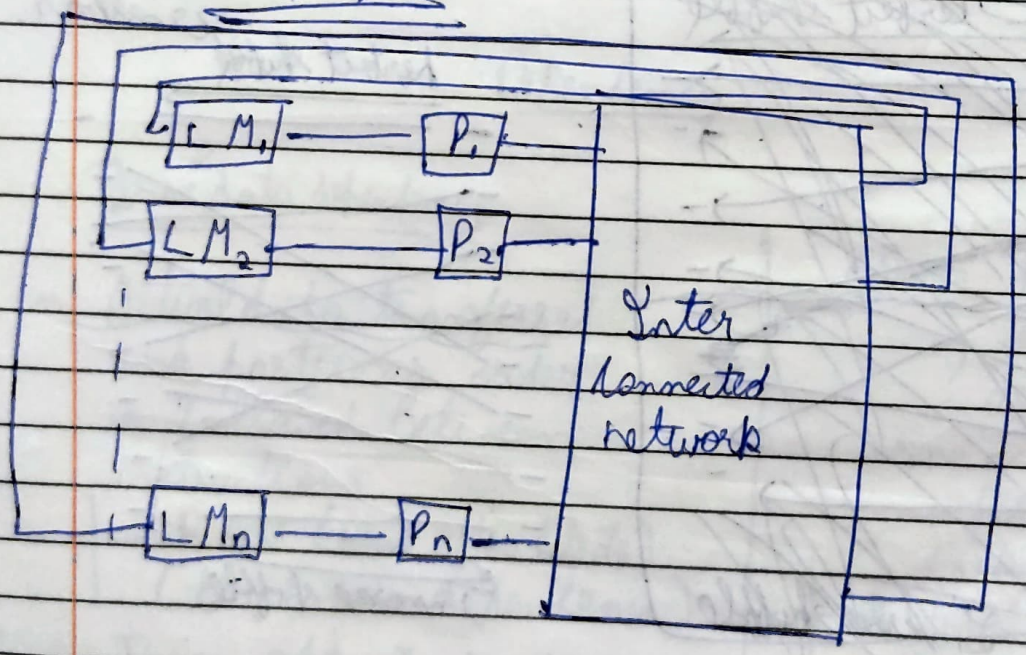
- → own local memory (least time)
- → shared memory (moderate time)
- → other processor's local memory (most time)

Date :
Page No.



(there can be individual cache memory of a processor)

UMA



(shared memory is also present)

NUMA

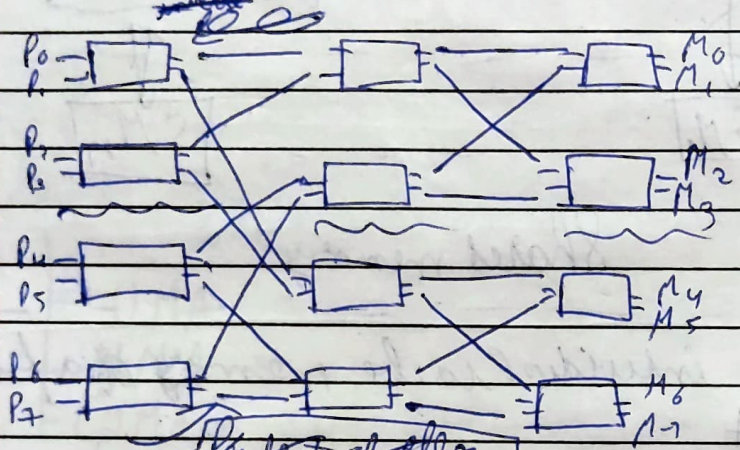
Inter connection network (not in syllabus)

- Blocking → Omega, Cube, Butterfly, Baseline
- re-arrangable → Benes
- non blocking → Clos

Date :
Page No.

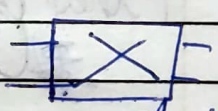
multistage network → made using software controlled switches (~~eg~~ simplest is a 2x2 switch)

Shuffle exchange network (8x8)



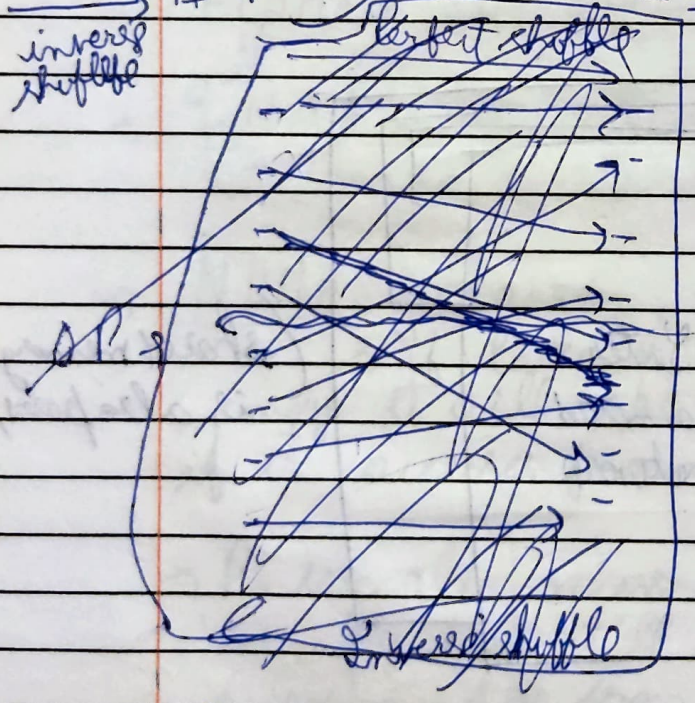
Switch configuration

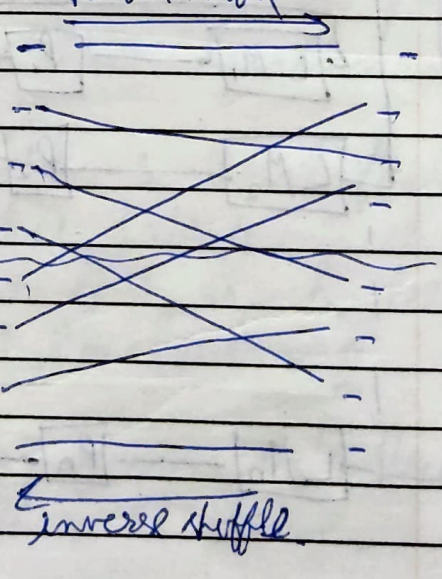
straight



crossbar

inverse shuffle





Baseline network

→ Recursive Kifs

Data dependency / Resource dependency

Date :
Page No.

Dependencies and Hazards :-

→ dependencies among processes / tasks / instructions
→ they are dependent upon input & output.
→ software feature (program)

e.g.:- I_1 : add r_1, r_2, r_3 // $r_1 \leftarrow r_2 + r_3$

(r_1: destination operand; r_2, r_3: source operands)

I_2 : mul r_4, r_5, r_6 // $r_4 \leftarrow r_5 \times r_6$

→ no dependency between I_1 & I_2 whether register/operand/source/resource etc.

Dependency

True data dependency

→ Actual data transferred and particular order is followed between instructions
- RAW (Read After Write) (true data dependency)

e.g.:- I_1 : add r_1, r_2, r_3

I_2 : mul r_4, r_1, r_6

if somehow I_1 is not completed, then I_2 will not execute. First write in R_1 in I_1 & then read in R_1 in I_2 hence called RAW.

False dependency

→ also called name dependency
→ no actual data is transferred between diff tasks
- WAW (Write after write) (output dependency)
- WAR (Write after Read) (Anti dependency)

Date :
Page No.

WAW eg:- I_1 : add r_1, r_2, r_3
I_2 : mul r_1, r_5, r_6

both instructions store the result at same location or register. If I_1 is executed before I_2 then its alright otherwise it will give wrong result. So write in r_2 and then write r_2 hence WAW. No actual data transfer between instructions hence false dependency.

⇒ if we change r_1 in I_2 to r_7 by using a free register or name register then the problem is solved. As only name is common hence called name dependency. This is called register renaming.

WAR eg:- I_1 : add r_1, r_2, r_3

I_2 : mul r_2, r_5, r_6

⇒ If an instruction later is writing ~~and~~ in a register and that register is ~~has~~ read before in an instruction earlier hence WAR.

⇒ If the order is changed then it will affect the result.

Date :
Page No.

Hazard

→ A hardware feature.
→ ~~There is a re~~
→ Dependency may be present but may not result in hazard.

Pipeline execution pattern

→ add r_1, r_2, r_3 // $r_1 \leftarrow r_2 + r_3$
mul $r_5, 10(r_1), r_6$ // $r_5 \leftarrow M[10 + r_1] * r_6$

r_1 required here

→ r_1 available at this moment

IF	ID	AG	DF	EX		
	IF	ID	AG	DF	EX	
		IF	ID	AG	DF	EX

as r_1 was not present at required time, it would wait for 2 unit cycles.

⇒ So this way pipeline does not work properly also called stall.
⇒ Hazard hence is also called pipeline stall, it is a situation in which pipeline gets stalled.
→ Arrises due to dependencies
⇒ This situation was RAW dependency.
→ WAW will work properly until we do not go out of order.
⇒ To avoid RAW dependency, we can reorder further instructions which will not have dependency in between I_1 & I_2.

Date :
Page No.

⇒ if we get 2 inst without dependency then the ~~ex~~ dependency time will ~~to~~ reduce by 2 units. It may not remove dependency, but reduce its effect. It is called Instruction reordering.

⇒ Policies for dependency handling :-

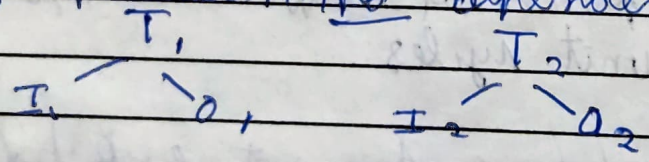
RAW ⇒ Instruction reordering
WAW ⇒ Register renaming
WAR

⇒ We can only increase performance upto a limit using pipelining. limits are latch delay

Bernstein's conditions for parallelism

⇒ ~~I~~ The code used should use the parallelism used.

⇒ for parallelism no dependency should be present.



(T → task, I → input, O → output)

True data dependency ① RAW : $I_1 \cap O_2 = \phi$

output dependency ② WAW : $O_1 \cap O_2 = \phi$

Anti dependency ③ WAR : $O_1 \cap I_2 = \phi$

if these 3 conditions are met, then the two tasks/processes are ~~called~~ said to be parallelizable

Date :
Page No.

Dependency Graph :- RAW →
WAW —o→
WAR —+→

Dependencies
- Data
- Control
- Resource

Control dependency :- It ~~ca~~ occurs because of program control instructions.

Branches
- conditional → ~~if else~~ JNZ, JZ, jump is not zero etc.
- unconditional → jump, call, RET (return)

eg ⇒

```
C = 10
do {
    C = C - 1;
} while (C != 0);
b = b + 1;
```

8085 code ⇒

```
      MVI C, 0A
LOOP: DCR C
      CMP C, 00H
      JNZ LOOP
      INR B
```

⇒ control instructions may lead to control hazards.
⇒ we may not get results for several pipeline cycles. it is similar to pipeline stalling.

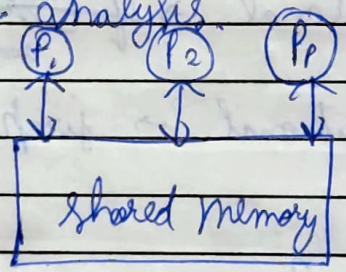
Handling control dependency :-

⇒ **Branch predictor :-** works on branch history (that may be saved in a history table)

Date :
Page No.

PRAM (Parallel Random Access Machine)

- A Theoretical model used for analysis of parallel algorithms
- a machine of P general purpose processor
- each processor may have its own local memory (cache/RAM/registers) but all communication between processors takes place through shared memory.
- Each processor knows its processor-id.
- All processors are synchronized to read, execute & write at the same time by one control unit. However, PRAMs are impractical & are the tools for algorithmic analysis.



- It can be ~~and~~ exclusive/concurrent operations, based on it, it can be divided into 4 parts:

- EREW → Exclusive Read Exclusive write
- CREW → Concurrent Read Exclusive Write
- ERCW → Exclusive Read Concurrent Write
- CRCW → Concurrent Read Concurrent Write.

EREW → At any moment at any memory location only one processor can read & one processor can write

Date :
Page No.

CREW → At any moment at any memory location, multiple read by processors & single processor can write.
Vice versa for other 2 -

→ Concurrent write leads to contention because multiple processors try to write at same location.
It requires special way to resolve the conflict.

- Random :- random value is written from data
- Arbitrary :- any value even out of data is written
- Priority :- A priority order is decided on some values and they follow this order in rotation (like round robin) to write.
- Combining

→ CRCW → take O(1) time from O(n) processors resources.

→ EREW → take O(n) time " O(1) " "

→ total work time complexity is same for both.

→ CREW → take O(logn) time from O(n) processors
work time complexity → O(nlogn)

(Study Binarytree Paradigm)

(Broadcast algorithm / Reduction algorithm) → Read

Date :
Page No.

Parallel sum of n elements

8 elements :-

A (0)

A(0) ——— A(1)
P_1

A(0) A(1) ——— P_0 A(2) A3 ——— P_1

A(0) A(1) ——— P_0 A(2) A(3) ——— P_1 A(4) A(5) ——— P_2 A(6) A(7) ——— P_3

Algorithm : sum

Input : Array A(1 : n) where $n = 2^k$
output : the sum of the values of array stored in A(1)
(slight differnce where array indexing starting from 1)

```
BEGIN
    P = n/2
    while P > 0 do
        For i = 1 to P do in parallel
            A(i) = A(2i-1) + A(2i)
        End parallel
        P = P/2
    End while
END
```

→ how to optimize this study this also

lab assignment : → Create multiple processes fork()
→ Implement shared memory based inter processor communication mechanism (system V IPC), shmget, shmctl, ftok, shmat, shmdt.

↳ Google ⇒ C, UNIX, DAVE, Beej's Guide.

Algorithm Optimal Sum :- ERCW

this can maybe wrong

Input :- An array of numbers $A(1:n)$

Output : The sum of these numbers in a variable called sum.

1.) For $i = 1$ to $\frac{n}{\log n}$ do in parallel

2.) using the sequential method find the sum of : $A(i-1)\log_{n+1} A_{(i-1)}\log_{n+2}$ ---- $A_i \log n$ and store result in variable B_i.

3.) End parallel.

4.) Find the sum of $B_1, B_2, \ldots B_{n/\log n}$ and store in variable sum

5.) Find optimal sum.

Group 1 : $A_1\ A_2\ A_3$ ---- $A_{\log n}$

Group 2 : $A_{\log n+1}\ A_{\log n+2}\ A_{\log n+3}$ ---- $A_{2\log n}$

Group 3 : $A_{2\log n+1}\ A_{\log n+3}\ A_{\log n+3}$ ---- $A_{3\log n}$

Date :
Page No.

Time to execute a program :-

→ Instruction count × CPI × cycle time

cycle per instructions

→ To improve performance → time to execute ↓ ⇒ The three parameters should get decreased by various means

Affecting Factors :-

→ Instruction set architecture → RISC : load/store architecture
— meaning they can be accessed using load/store instructions only
RISC instructions run faster individually

→ CISC → ALU instruction can have memory as one of the source operand
→ Instructions are complex, require more time in decoding, so run slower.

$x = Y + Z$

RISC
```
load R1, Y
load R2, Z
ADD R3, R1, R2
Store X, R3
```

CISC
```
MOV R1, Y
ADD X, R1, Z
```

↓

CPI × cycle time is reduced in RISC

Date :
Page No.

⇒ Pipelining : it affects only CPI, it
→ it'll be less in parallel as compared to sequential.
→ Improves time.
→ Too much pipelining can cause control hazards and after one limit they increase time.

⇒ Compiler Optimization

One eg. is instruction reordering.

→ reduces CPI
→ write good among all the above methods.

⇒ Better Fabrication Technology.

⇒ Parallel Architecture (Multicore processor)

⇒ Loop unrolling

① for (c=1; c!=100; c++)
{a[c] = a[c+2] + 10;
}

↓ unrolled

② for (c = 1; c!=100; c=c+2)
{a[c] = a[c+2] + 10;
a[c+1] = a[c+3] + 10;
}

```
MVI, 01H
LOOP: CMP C, 64H
      JZ EXIT
      Load
      ADD
      Store
      INCC
      JMP LOOP
EXIT:
```

Date :
Page No.

Q.

```
      MVI C,01H
LOOP: CMP C,64H
      JZ EXIT
      LOAD
      ADD
      STORE
      LOAD
      ADD
      STORE
      ADI C,02H
      JMP LOOP
EXIT:
```

These loop overhead instructions executes 50 times whereas 100 in prev code, so it reduces drastically.

⇒ CPI reduces
⇒ too much of loop unrolling can lead to cache misses.

Granularity (Grain Size) & Communication Overheads

Grain Size → amount of computations involved in a software process (no. of instructions count of instns)

- Fine grain → Instn level < 20
  - → loop level < 500
- medium grain → Subroutine < 200
  - → Subprogram (thousand of instructions)
- coarse grain → Job / Program level (tens of thousands of instructions)

Amount of parallelism increases However communication overheads increases

Date :
Page No.

program
- → Instruction level
- → loop level
- → subroutine (functions)
- → threads
- → complete program

Communication overheads → latency → is a time measure of computation overheads

→ Memory latency : time required by processor to access memory

→ Synchronization latency : time required for two processors to synchronize with each other.

Hardware Resources ⇒ compiler
program developer
operating system

⇒ at fine grain level we exploit parallelism using compiler / threads.

→ at medium level ⟶ program developer
→ at course level ⟶ operating system

Date :
Page No.

e.g. ⇒

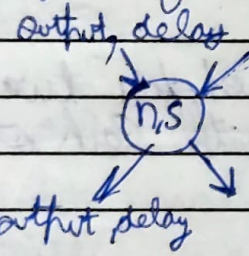
1. a = 1
2. b = 2
3. c = 3
4. d = 4
5. e = 5
6. f = 6
7. g = a × b
8. h = c × d
9. i = d × e

grain size = 1 due to only 1 operation (=)

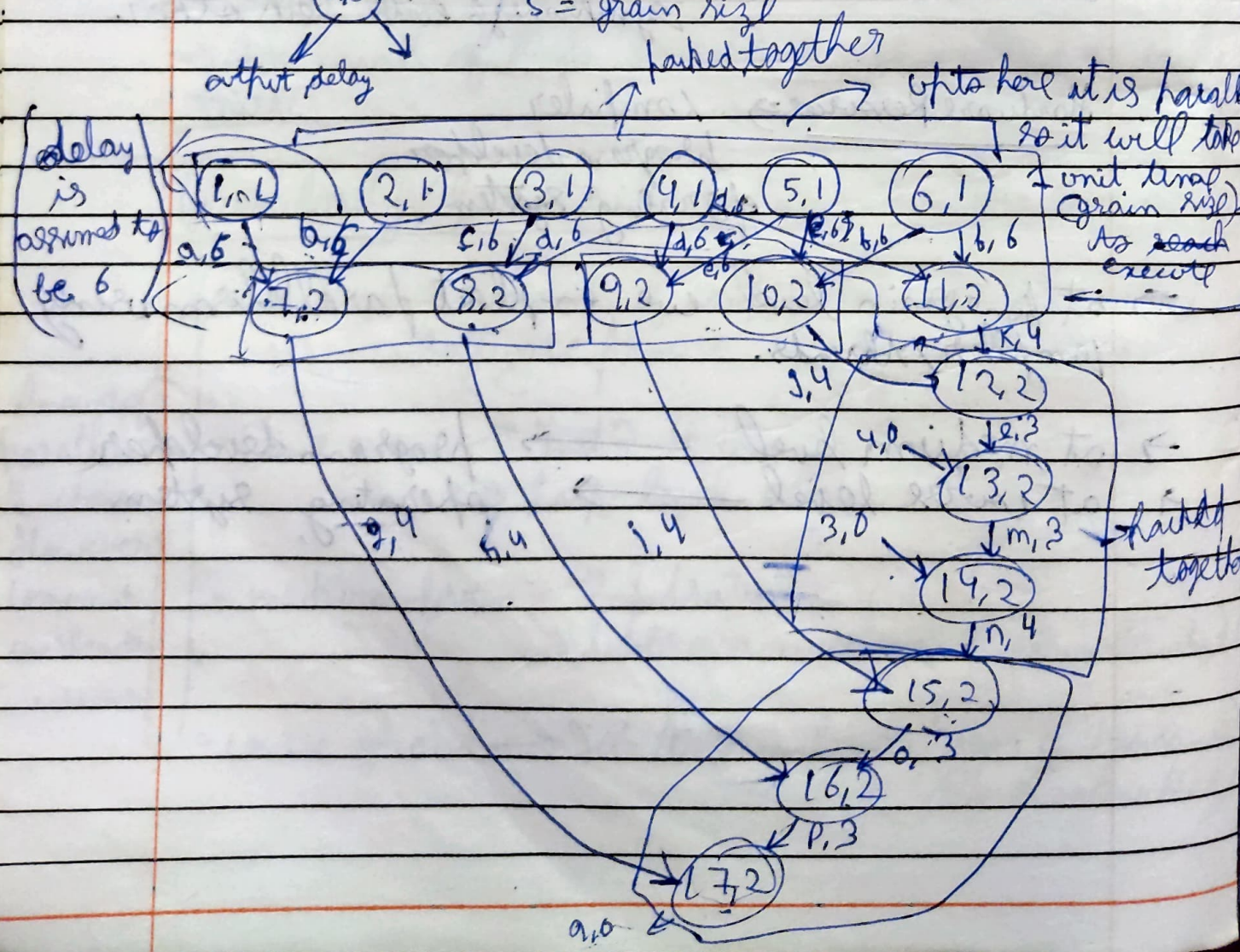
10. j = e × f
11. k = d × f
12. l = j × k
13. m = 4 × l
14. n = 3 × m
15. o = n × i
16. p = o × h
17. q = p × g

grain size = 2, due to only 2 operation (=, ×)

⇒ program flow graph of above at fine grain level.



n = node number
s = grain size



Date :
Page No.

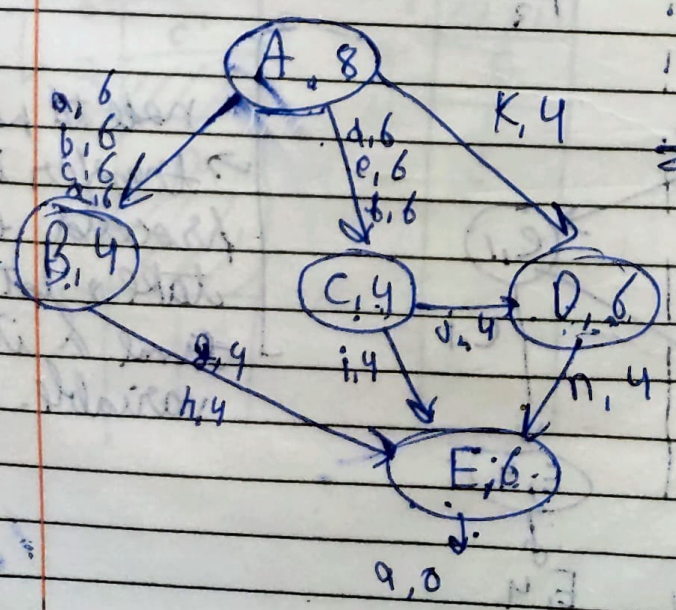
→ after it, it will take 6 units time to reach (11) node

→ after it ~~will~~ we'll calculate total time by adding delays & grain size.

→ when we reach node (15), we have already got result from (9) node so no extra delay there and same with the cases for nodes (16) & (17). So final will be 41 units (calculate it)

⇒ in sequential system (single processor system) inter node communication delays may be neglected. So it comes out to be 28 units (calculate it) in sequential. So it proves that at finer grain level, we may increase parallelism, but increase communication overhead, which may hamper the time required.

→ Grain packing :- pack more than 1 node together and sum their grain size.



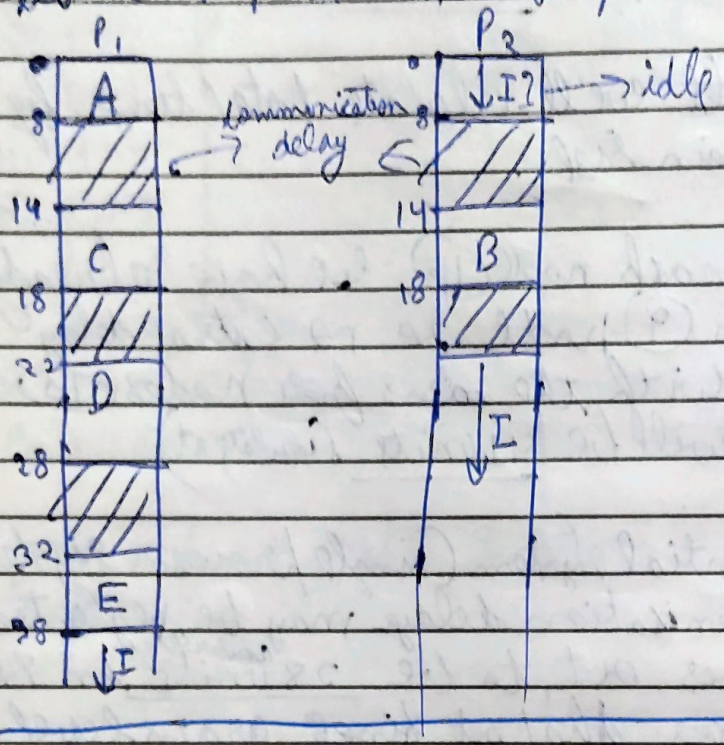
A → B, C, D C → D E
8 + 6 + 4 + 4 + 6 + 4 + 6
B, C D B, C, D → E

= 38 units.

⇒ ultimately sequentiality decides how much time it will take.

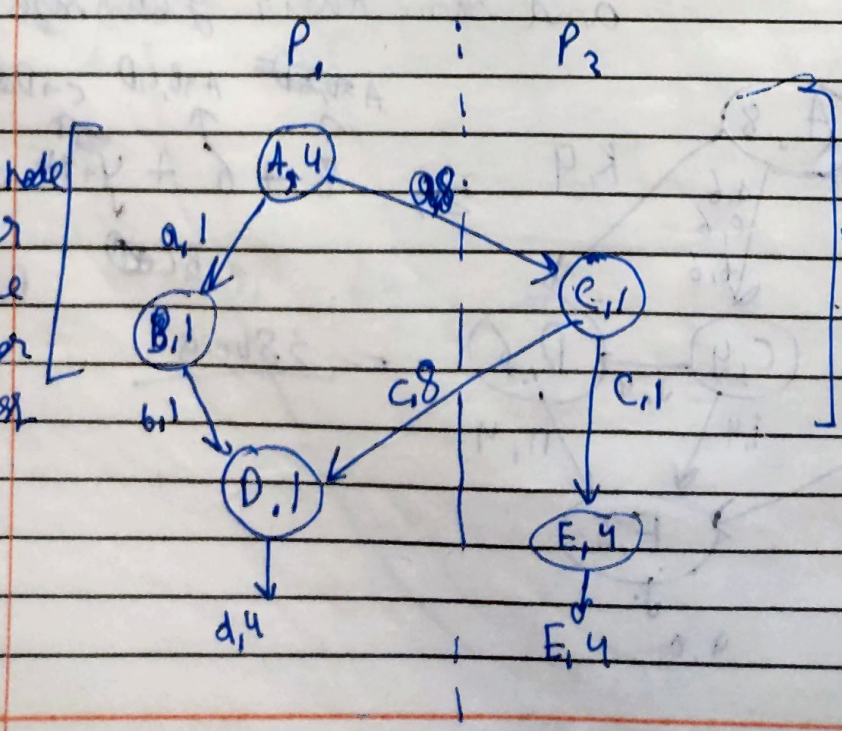
Date :
Page No.

Above PFG in two processor system :-



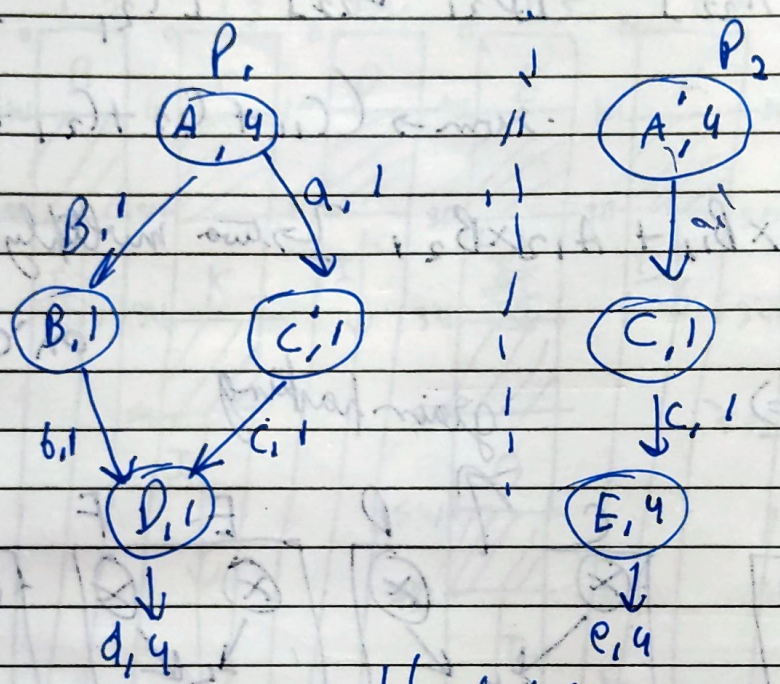
Node Duplication in Static Multiprocessor Scheduling

⇒ we need to exploit just optimal level of parallelism.

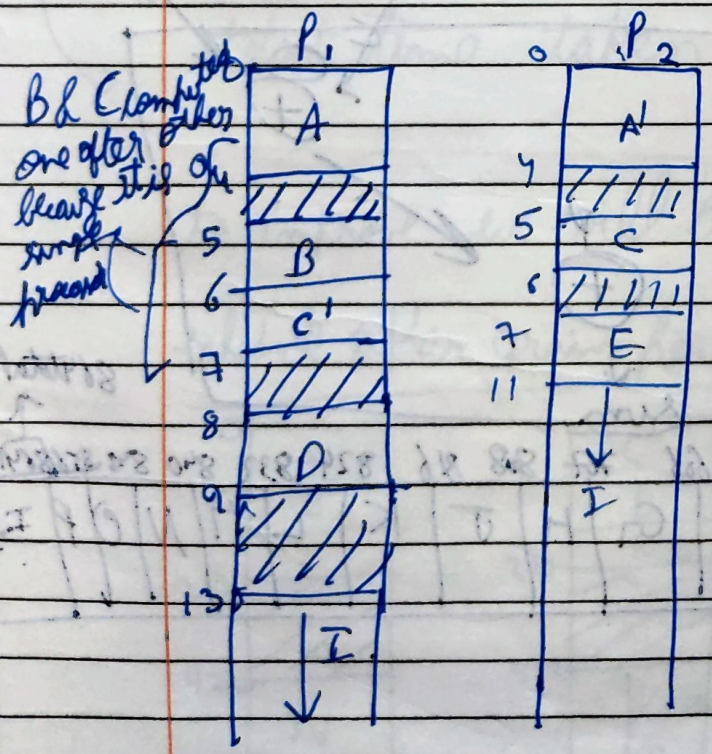


Date :
Page No.

or to decrease the node to node communication b/w different processors we duplicate the nodes.



⇓ schedule



B & C computed after one after because its on single processor

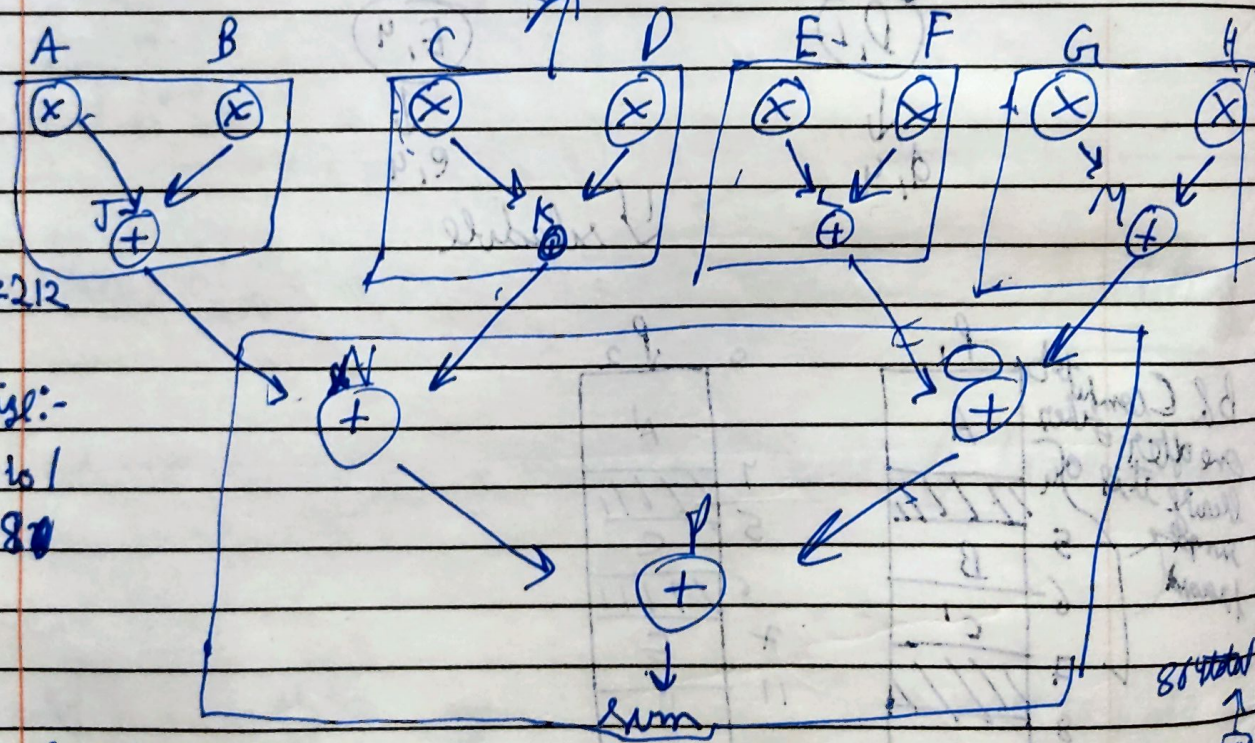
Date :
Page No.

e.g.) $\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{21} \end{bmatrix}$

sum → $(C_{11} + C_{12} + C_{21} + C_{22})$

$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$ → two multiply & one add in calculating on 'C' value

DFG ⇒ grain packing



delay d=212

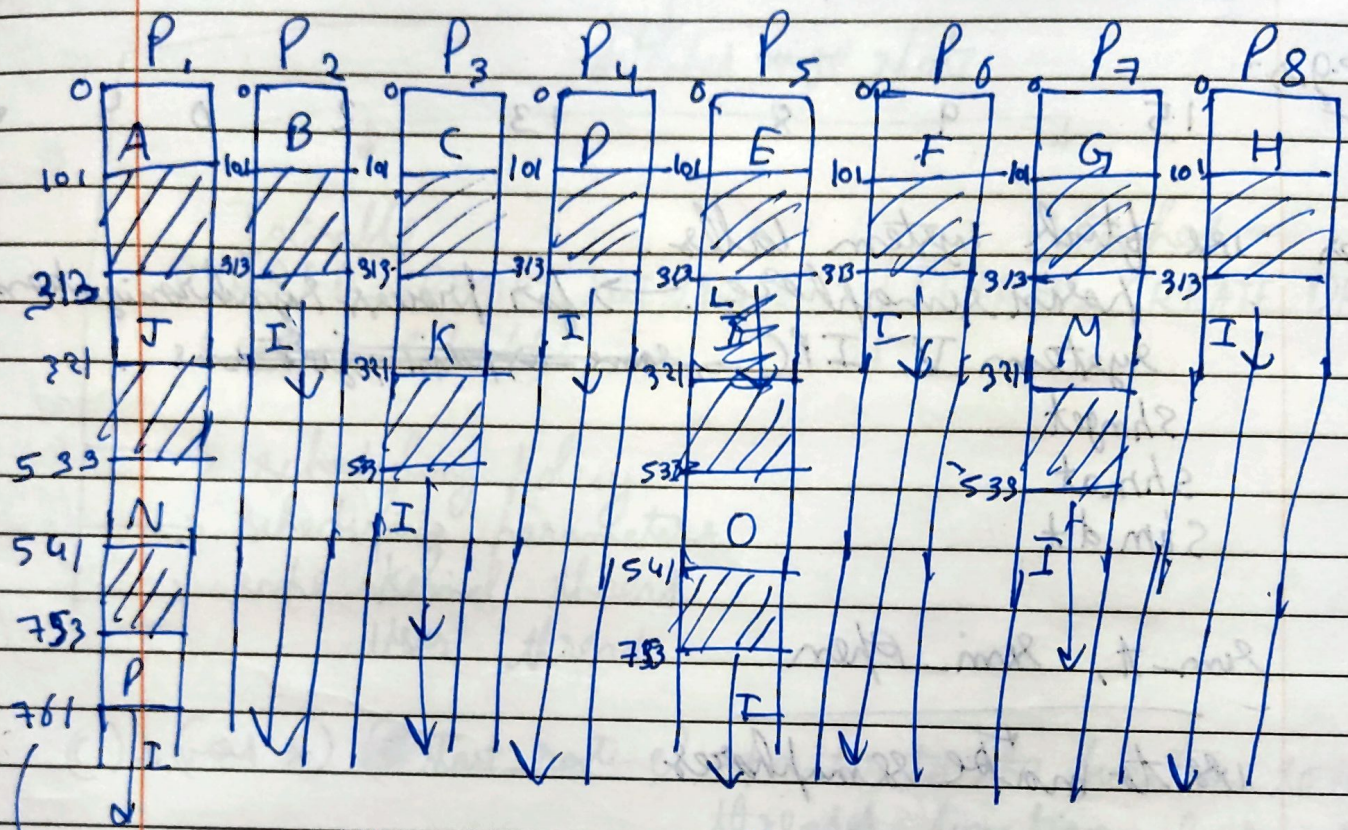
grain size:-
⊗ → 101
⊕ → 8

Sequential schedule ⇒

101	202	303	404	505	606	707	808	816	824	832	840	848	856	864	872
A	B	C	D	E	F	G	H	J	K	L	M	N	O	P	I

Date :
Page No.

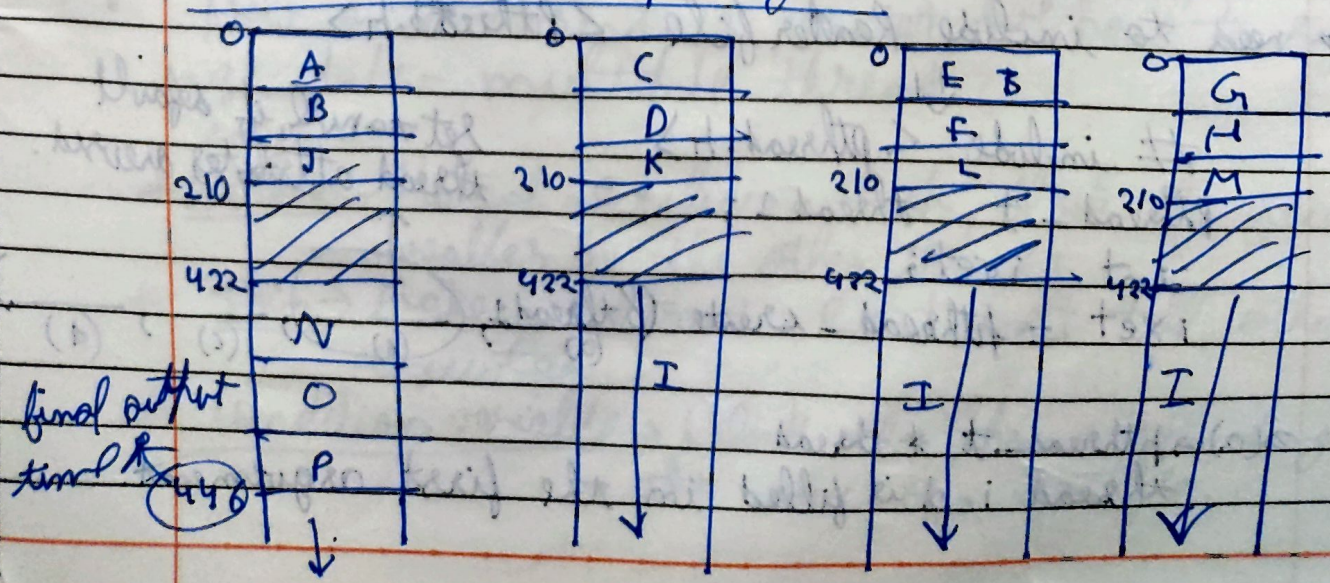
now if we take max parallelism as 8 :-



total time taken → $\boxed{761}$, i.e., not significant improvement.

to improve we will do grain packing

schedule after grain packing :-



Date :
Page No.

Odd-Even transposition Sort using multiple process

e.g.:-

15	9	8	13	2	0	4	1

⇒ use fork system calls.
posix semaphore → for process synchronization
system V IPC → ~~same as posix, just one~~
shmget
shmat
shmdt

sem_t, sem_open

use to make semaphores

sem_wait
sem_post

Posix Threads (Pthread)

⇒ need to include header file <pthread.h>

```
#include <pthread.h>
pthread_t thread1;
int iret;
iret = pthread_create (&thread1, ____, ____, ____)
```

(a) (b) (c) (d)

set to null, if default thread attributes are used.

⇒ (a) → pthread_t * thread
thread id is filled in the first argument.

Date :
Page No.

(b) → we specify the attributes of thread being created

detached ~~state~~ state

Joinable	Detached
PTHRED_CREATE_JOINABLE (set as default)	PTHRED_CREATE_DETACHED

→ scheduling policy
→ scheduling parameters
→ scope : Kernel thread
User thread

(c) → (void *) (start_routine) :- pointer to the function to ~~the~~ be threaded, function has a single argument - pointer to void.

(d) → void * arg :- pointer to argument of function to pass multiple arguments send a pointer to structure.

- Now once thread is created, we have to synchronize b/w multiple thread

ways :-

* Mutex → Mutual exclusion lock, block the access to variables by the other threads
* Join → Make a thread wait till other are executed
* Condition variable → Block the thread depending upon the values of a variable.

Date :
Page No.

Mutexes → pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIATE;

```
pthread_mutex_lock (&mutex1);
// critical section
pthread_mutex_unlock(&mutex1);
```

Join → pthread_Join(pthread_t threadid, NULL)

pthread_exit ( ); [if the main thread ~~does not~~ get executed it will exit & other threads will still remain active]

e.g.→

```
#include <stdio.h>
#include <pthread.h>
# define NTHREADS 10
void * thread_function (void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;
int main()
{
    pthread_t thread_id[NTHREADS];
    int i, j;
    for (i=0; i <NTHREADS; i++)
    { pthread_create &thread_id[i], NULL, thread_function, NULL
    }
    for (j=0; j < NTHREADS; j++){
        pthread_join (thread_id[j], NULL);
    }
```

this will create threads

Date :
Page No.

```
print & ("Final counter value is %d \n", counter);
return 0;
}
void * thread_function (void * dummy)
{ print & ("Thread number %d \n", pthread_self());
  pthread_mutex_lock (& mutex1);
  counter++;
  pthread_mutex_~~lock~~unlock (& mutex1);
}
```

MPI → Message Passing Interface (used for point to point communication)

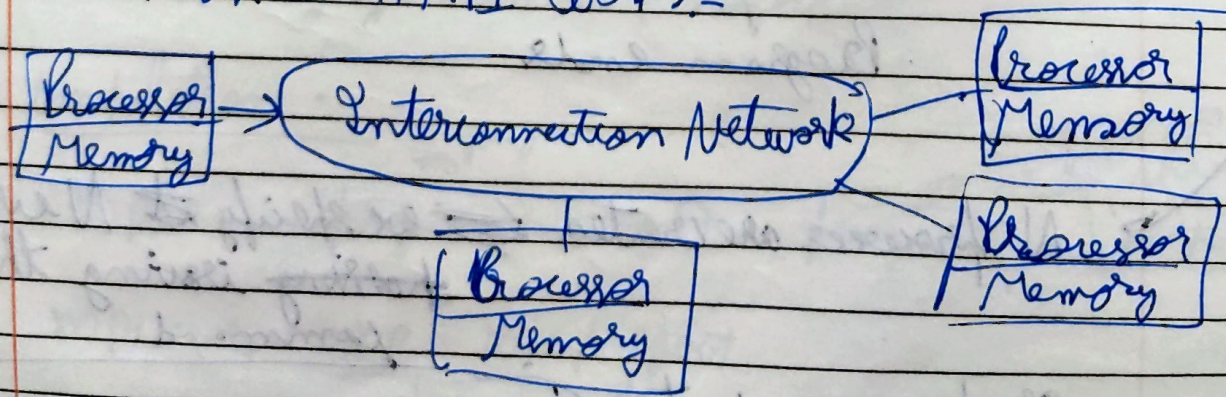
→ Initially it was developed for distributed Memory system, but presently these are available for shared memory ~~&~~ system also. (hybrid too)

→ Need for MPI:-

→ Initially each ~~&~~ vendor had different parallel program, ~~&~~ due to this portability issues arrised.

→ So, a common interface had to be developed.

→ Model on which MPI works:-



→ user specifies the number of concurrent processes when the program begins and typically the number of active processes remains constant throughout the execution

Date :
Page No.

of the program.

→ Every process executes the same program but each one has a unique ID number, because of which different processes may perform different separations as the program unfolds.

→ A process alternatively performs computations on its local variable & communication with other processes or i/o devices.

MPI Include File # include <mpi.h>
Declarations, prototypes
Program begins
serial code
Initialize MPI Environment (parallel code begins)
Do work & make message passing calls
Terminate MPI Environment (parallel code ends)
serial code
Program ends

→ N processes are created ← we specify N while ~~typing~~ issuing the execute command.

group communicator MPI_COMM_WORLD (Default)

initialization of MPI environment

Date :
Page No.

⇒ MPI_INIT (&argc, &argv)

null may also be passed

⇒ if the function call is not valid or the return value is not valid then the process will get aborted

MPI_Finalize (); ⇒ to end MPI environment

⇒ It may not be present at beginning of main but untill it is not executed, processes are not created & the function executes serially.

⇒ MPI_COMM_rank (MPI_COMM_WORLD, &pid);

⇒ in a communicator if N processes are created then the rank (pid) will go from 0 to N-1.

⇒ MPI_Comm_size (MPI_COMM_WORLD, &wsize);
communicator name
↳ this way we will get how many processes are there.

⇒ compilation :-

mpicc prog.c -o progout
program name
if we want to change name of output file

mpirun -np 10 progout
↓
no. of concurrent processes to be created.

MPI_Abort (comm, errorcode)] → aborts all MPI calls associated with communicator

Date :
Page No.

MPI_initialized (& flag) → to check if MPI_Init is called.

MPI_Wtime → returns an elapsed wall clock time in seconds (double precision) on the calling processor.
[double MPI_Wtime(void)]

MPI_Wtick → returns the resolution in seconds (double precision) of MPI_Wtime.
[double MPI_Wtick(void)]

MPI_finalize → no other MPI routine may be called after it.

⇒ we can compute time of a section of code by putting a pair of calls to function MPI_Wtime before & after the section.

⇒ practically the processes created do not start executing the process together.

→ The difference between two values returned by the function is the number of seconds elapsed.

→ From a logical point of view, every MPI process must begin execution at the same time, but this not in pratice.

→ MPI_Barrier (to start all the processes together)
int MPI_Barrier(MPI_comm comm)

after this every process will wait until each every process has not executed this ~~process~~ function

Date :
Page No.

⇒ function calls → point to point
→ Broadcast

⇒ Point to point Communication Routines
(Blocking Message passing routines)

MPI_Send , MPI_Recv [more processes available but we are only studying these 2]

(for sending) MPI_Send prototype : MPI_Send (
void * data,
int count,
MPI_Datatype datatype,
int destination
int tag
MPI_Comm comm)

[Buffer containing the information to be sent]

[How much (count of data) of datatype is present in Buffer]

MPI_datatype ⇒ MPI has its own datatype in C language.

⇒ MPI_CHAR ⇒ unsigned char
MPI_INT ⇒ int
MPI_UNSIGNED_CHAR ⇒ unsigned char

⇒ similar corresponding data types for double, float etc.

⇒ int destination ⇒ rank (processid) of process to which data is to be sent.

⇒ int tag → message id / message type
↳ helps in sending & recieving blocking.

Date :
Page No.

for example, if sender has tag = 1 and receiver has tag = 2 then it will get stored in buffer until it does not get message with tag = 1. When it receives tag = 1, blocking will stop.

MPI_Recv prototype :- (for receiving)

```
MPI_Recv (
    void * data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag; ---> MPI_ANY_TAG (predefined constant)
    MPI_Comm comm,
    MPI_Status * status )
```

extra attribute added.

Example code :-

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main (int argc, char * argv)
{
    MPI_INIT (NULL, NULL)
    int wrank;
MPI_Comm_rank (MPI_COMM_WORLD, & wrank);
    int wsize;
MPI_Comm_size (MPI_COMM_WORLD, & wsize);
    int number;
```

Date :
Page No.

```
if (wrank==0)
  {  number= -1;
     MPI_Send (&number, 1, MPI_INT, 1,0, MPI_COMM_WORLD);
  }
else if (wrant==1)
  {
    MPI_Recv(&number, 1, MPI_INT, 0,0, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);

printf ("Process 1 ~~required~~ recieved number %d from
          process 0\n", number);
  }
  MPI_finalize();
}
```

Note: MPI_Send (first word capital then small)

MPI_Recv

- ~~Bsend~~
- ~~Sendrecv~~
- Wait
- Waitany
- Waitall
- Waitsome
- Probe
- Get_count

Synchronous

MPI Blocking message Routines

Non blocking :-

MPI_Isend
- Irecv
- Issend
- Test
- Testany
- Testall
- Testsome
- ~~Test~~ Iprobe

(some part will be available in notes shared)

Date :
Page No.

Collective Communication:-

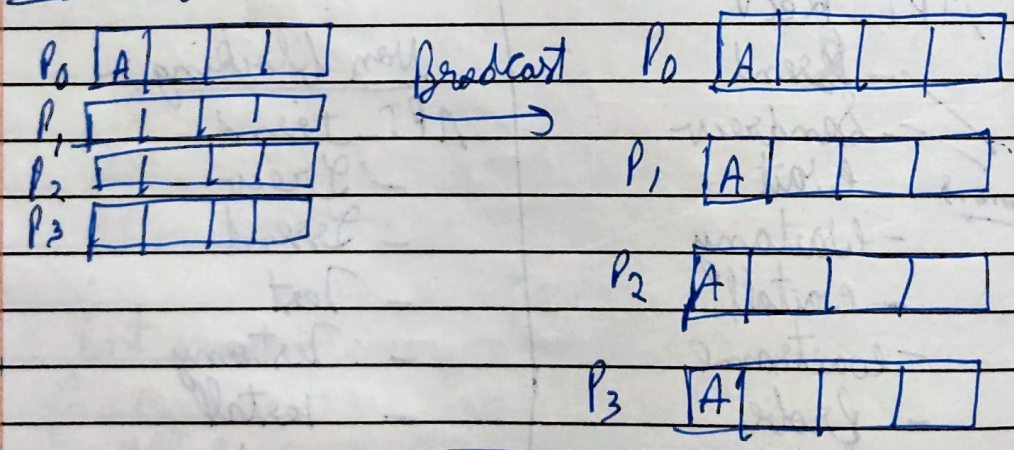
3 purposes:-

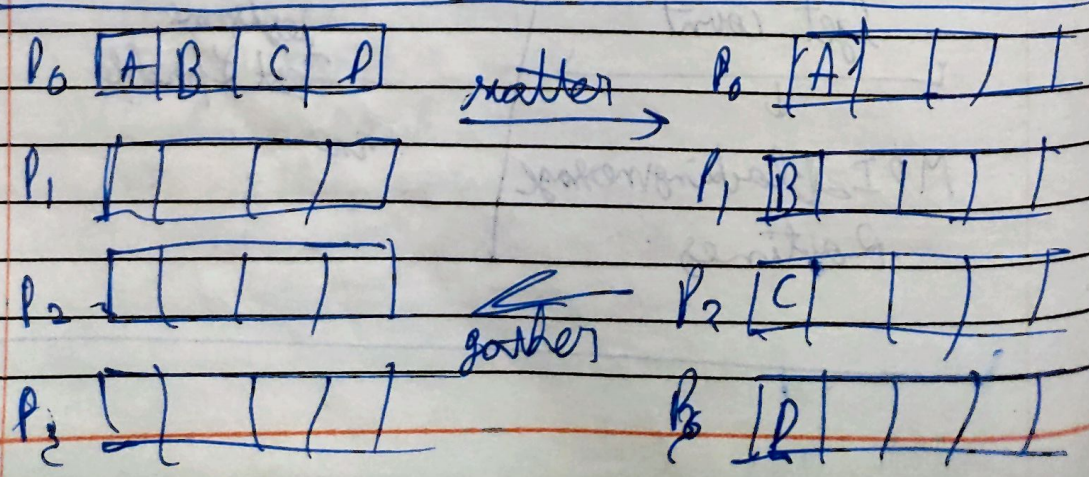
→ Synchronization — MPI_Barrier
→ Data movement — MPI_Bcast
→ Collective Computation — MPI_Reduce

MPI_BCast (
void * data, to_be_sent,
int send_count,
MPI_Datatype datatype,
int broadcasting_process_id,
MPI_Comm Comm);

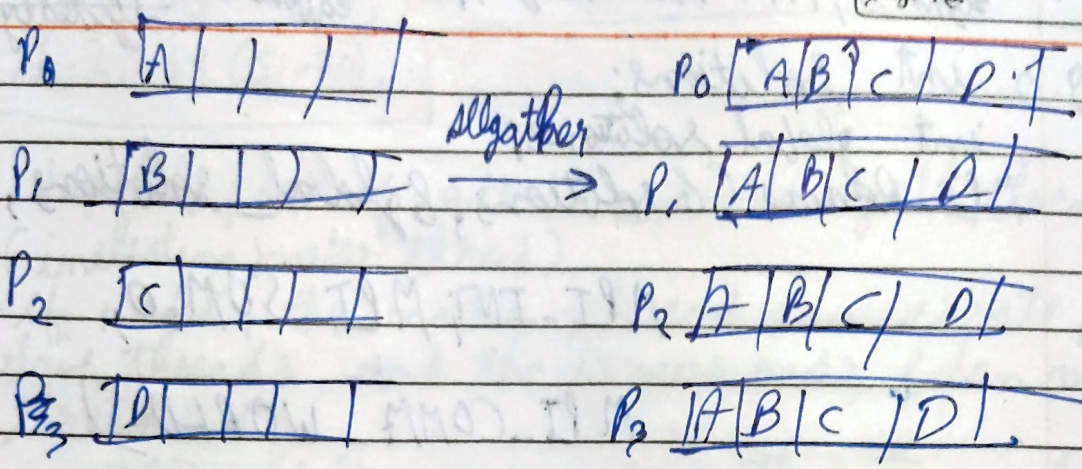
→ input parameters

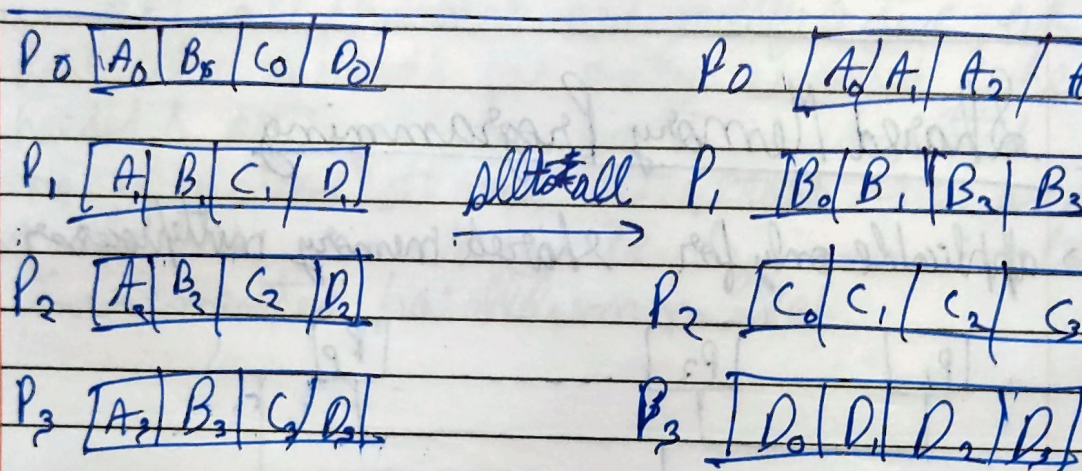
Broadcasting ~~example~~ operations :-





Date :
Page No.





→ int MPI_Reduce (
void * operand,
void * result,
int count,
MPI_Datatype datatype,
MPI_Op operator, → there are predefined operators :-
int root,
MPI_Comm comm);

- MPI_BAND
- BOR
- BXOR
} Bitwise

- LAND
- LOR
- LXOR
} logical

- MXM
- MAXLOC
- MIN
- MINLOC
- PROD
- SOM

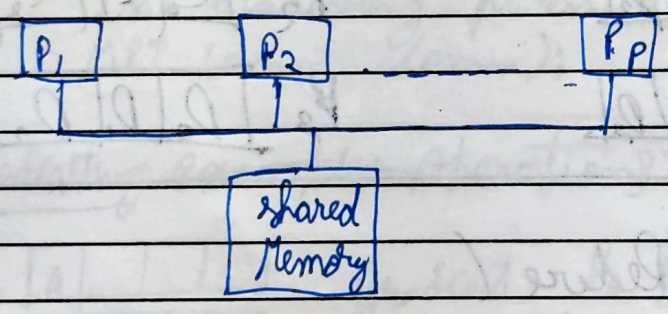
chapter 1 & 2, book exercise

* Flynn's, PRAM model, Gustafson, processor performance equation (different parameters and how they are affected by attributes)

e.g.) int solutions;
int global_solutions;
MPI_Reduce (&solutions, &global_solutions, 1,
MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD)

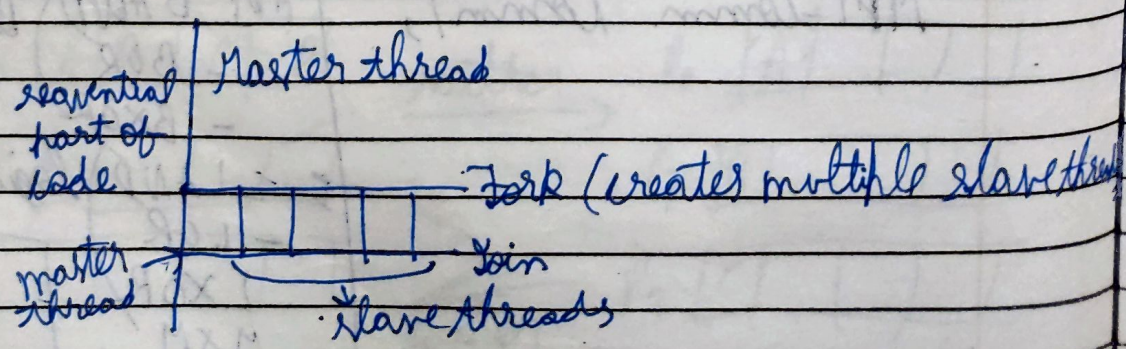
Shared Memory Programming

→ applicable only for shared memory multiprocessor



→ fork/join kind of parallelism model is used.

⇒ in shared memory multiprocessor threads are used for implementing parallelism, while in MPI, processes are used.



grain packing, processor schedule, node duplication.

ch-1, page 11-27, Book exercise

Date :
Page No.

ch-2, pg (51-70).

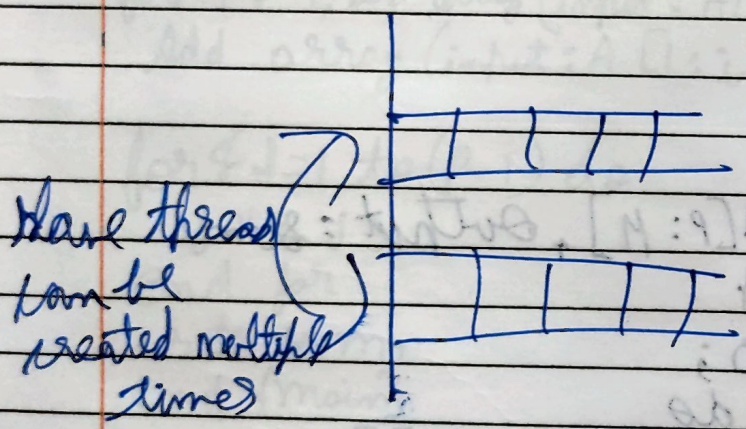
⇒ parallel part of code is executed by all threads (including master thread)

⇒ if needed, the master thread can create further slave threads, and the process ends when master thread ends.

⇒ in MPI all processes are mostly active, while here, the threads are only created when needed & destroyed when work is completed.

⇒ no. of active thread at start is 1 and also at the end, and can be dynamic in between.

⇒ Incremented parallelism model.



⇒ only where there is scope of parallelism we will use multithreads.

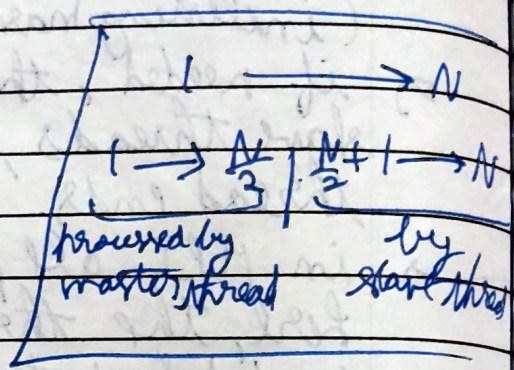
⇒ ~~Open~~ OpenMP is an API for programming in Fortran/C/C++ on shared memory parallel computers.

⇒ The API comprises three different components:-
- Compiler directives
- Library functions
- Environment variables

Date :
Page No.

→ Adding an array with 2 processor system (shared memory computer)

P_0 → master, P_1 → slave




```
BEGIN {main}
    global sum, Array A[1:N]
    for i:=1 to N
        Read A[i];
    end for
    sum:=0;
    fork(1) Add-array(input : A[N/2+1:N], output:sum);
    Add-array(input : A[1:N/2], Output : Sum);
    Join 1;
    write sum
    end {main}
```

will wait till thread 1 gets terminated

```
Add-array(input : A[P:M], output : sum)
BEGIN {Add-array}
    sum_local := 0;
    for i=P to M do
        sum_local := sum_local + A[i];
    end for;
    lock sum → (call mutex)
    sum:=sum + sum_local
    unlock sum
    end {Add-array}
```

Date :
Page No.

adding with P processor systems

```
BEGIN {main}
  global sum, array A[1:N]
  for i:=1 to N
     Read A[i];
  end for
  sum := 0;
  increment := N div P;
  for j=1 to(P-2) do
   fork (j) Add_array (input : A[j*(increment+1):(j+1)*increment],
                       output : sum);
  end for
  fork (P-1) Add_array (input : A[(P-1)*(increment+1) : N], Output : sum);
   Add_array (input : A [1 : increment], output : sum);

  for j=1 to (P-1) do
    join j;
  end for
  write sum;
  end {main}
```

this program is based upon SPMD (single process multiple data) programming.