Parallel And Distributed Computing

Category	
Files	
Created	@September 21, 2022 8:03 PM
Reminder	
Status	Open
URL	
Updated	@May 21, 2023 12:14 AM

▼ old stuff

Syllabus - Instruction Pipeline , branch prediction, branch history table, branch predictor, data dependence, Bernstein's condition, + tomasulo's algo

▼ Pipelining

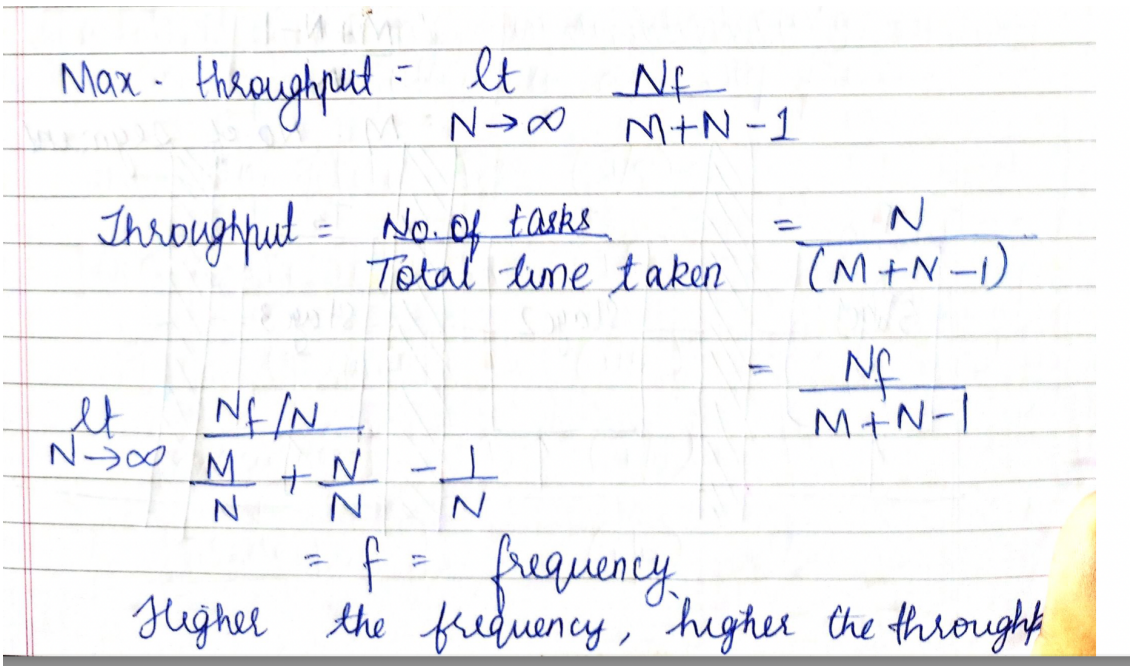
Instruction execution -

IF - ID - OF - Ex - Wb

Number of stages = M

Number of tasks = N

Total time taken for completing tasks = M + N - 1



Higher frequency, higher through put, higher power consumption, more heat generated

$$\text{speed up} = \frac{\text{Time taken without pipeline}}{\text{Time taken with pipeline}}$$

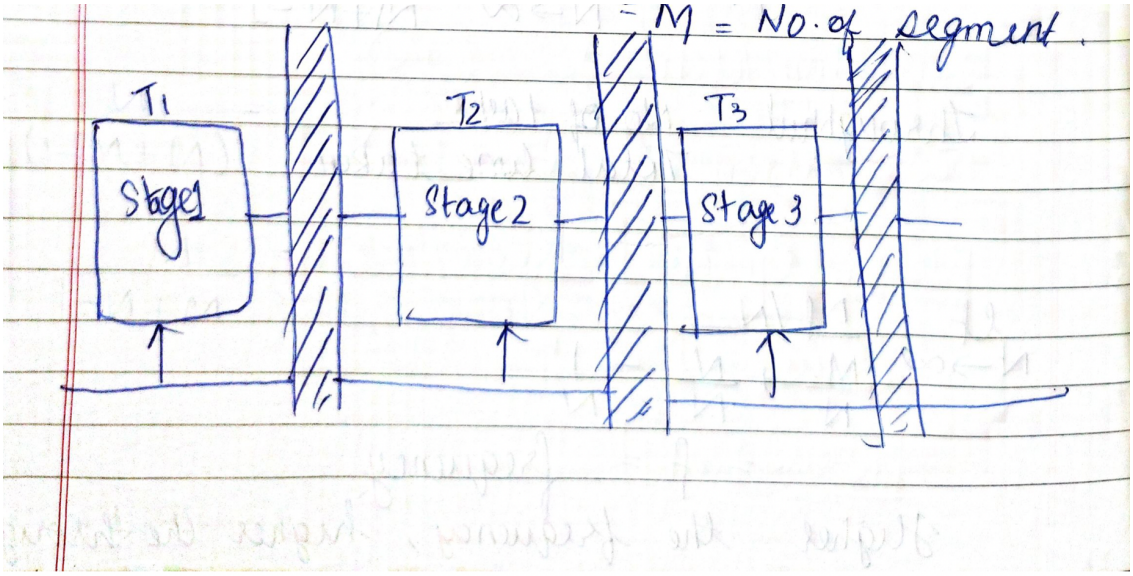
$$= \frac{MN}{M+N-1}$$

$$\text{Max speed up} = \lim_{N \to \infty} \frac{MN}{M+N-1}$$

$$\frac{\frac{MN}{N}}{\frac{M}{N} + \frac{N}{N} - \frac{1}{N}} = \frac{M}{\frac{M}{N} + 1 - \frac{1}{N}}$$

$$\frac{M}{\frac{M+N-1}{N}}$$

$$= M = \text{No. of segment.}$$

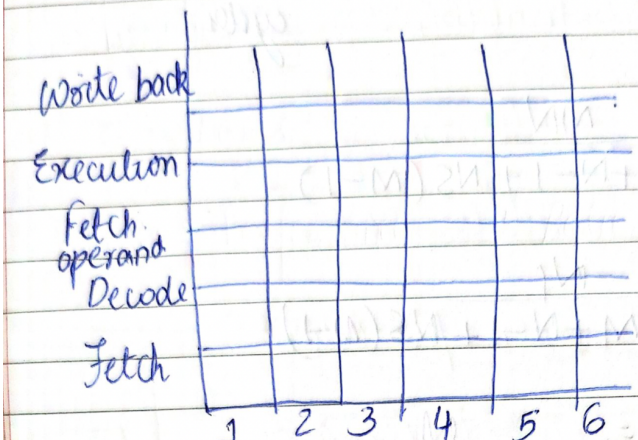


23/08/22

classmate
Date
Page

$$f = \frac{1}{\text{latch time} + \max(T_1, T_2, T_3, T_4)}$$

Instruction pipeline



No. of Instructions = N
No. of segments = M

Probability that a given Instruction will cause branching = S

Without branching = $m+n-1$

$= 5+15-1$

$= 20-1$

$= 19$

No. of extra clock cycles due to 1 branch instruction = $(m-1)$

$= 5-1 = 4$

Avg. no. of branching $= \sum_{i=1}^{N_i} 1 \cdot S + 0(i-S)$

$= NS$

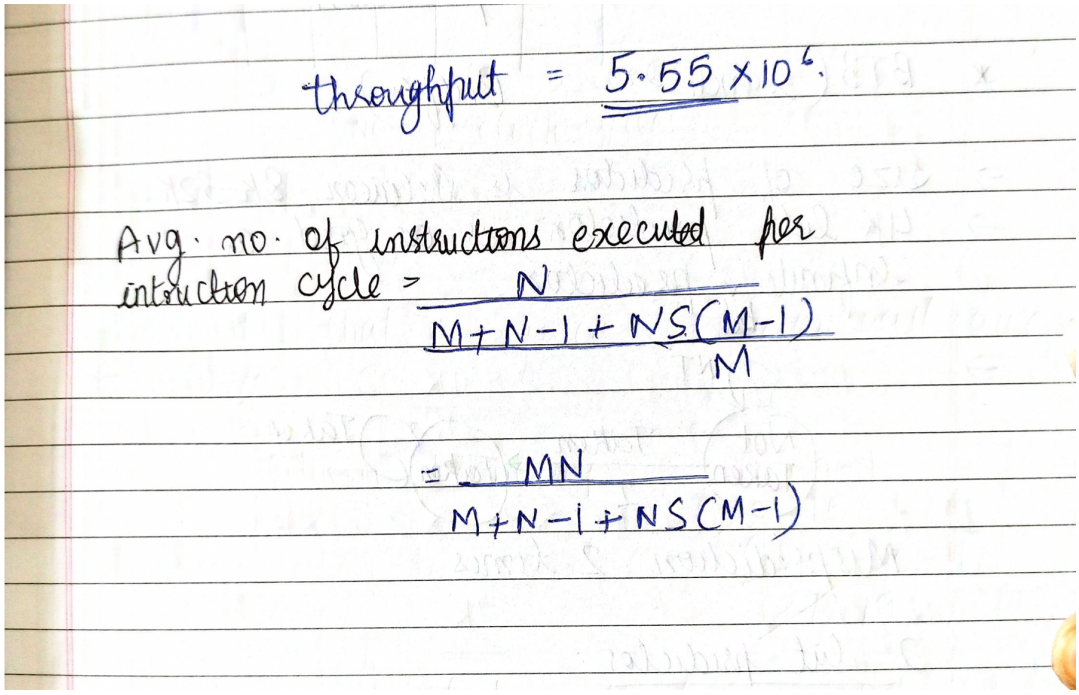
Time $= M+N-1+NS(M-1)$ clock cycles.

Speed up $= \frac{MN}{M+N-1+NS(M-1)}$

Throughput $= \frac{Nf}{M+N-1+NS(M-1)}$

Max. speed up $= \frac{M}{(M-1)S+1}$

Max. throughput $= \frac{f}{1+S(M-1)}$



https://www.youtube.com/watch?v=8Tdi2jKpISs&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=23&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=TEg_qFE50t0&list=PLsFENPUZBqiprGoEWrpexJPlPjmfPCtzM&index=44&ab_channel=GATEBOOKVIDEOLECTUR

https://www.youtube.com/watch?v=nZqYc5SAUxE&list=PLsFENPUZBqiprGoEWrpexJPlPjmfPCtzM&index=45&ab_channel=GATEBOOKVIDEOLECTU

https://www.youtube.com/watch?v=K6490LmUyNE&list=PL4yL5rqgtVtrj39MYMzq_LgER4Ijow8ld&index=29&ab_channel=MakeItEasy%23padhai

numerical -

https://www.youtube.com/watch?v=G7Tvb6hqXYQ&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=26&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=w02IamyWnFY&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=28&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=RQ5hHIwLAzg&ab_channel=EZCSE

https://www.youtube.com/watch?v=PBadU2XBGU0&ab_channel=RituKapurClasses

▼ Hazards and dependency

https://www.youtube.com/watch?v=snGZ64tc1l4&ab_channel=SuccessGATEway

https://www.youtube.com/watch?v=vEW70ruIbZ8&list=PLeWkeA7esB-PcOTrTCvAsaCArnCMQkcNv&index=3&ab_channel=Prof.Dr.BenH.Juurlink

https://www.youtube.com/watch?v=srlgaJgaxRE&ab_channel=GateSmashers

https://www.youtube.com/watch?v=HZNGbwFI4Rs&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=18&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=MKxbNhdhncE&list=PLGuh2K9TUN4Qxkrylyq_qwfhAZiuzZ2gz&index=22&ab_channel=AllAboutCSIT

https://www.youtube.com/watch?v=nC6csdXEkzU&list=PLYzc3veT5szJzLsUkeDnVQOsTnqCDzIFK&index=8&t=3s&ab_channel=RituKapurClasses

▼ Branch Prediction

https://www.youtube.com/watch?v=zdbyzJJCEbI&ab_channel=AllAboutCSIT

http://web.cecs.pdx.edu/~zeshan/ece586_lec17_1.pdf

https://www.youtube.com/watch?v=PFmx2p6NA0A&list=PLeWkeA7esB-PcOTrTCvAsaCArnCMQkcNv&index=8&ab_channel=Prof.Dr.BenH.Juurlink

https://www.youtube.com/watch?v=obUTjupcuBM&list=PLsFENPUZBqiprGoEWrpexJPlPjmfPCtzM&index=83&ab_channel=GATEBOOKVIDEOLECTUR

https://www.youtube.com/watch?v=4mXreTehXUw&ab_channel=TheVertex

https://www.youtube.com/watch?v=RShaZENRGFg&ab_channel=PadraicEdgington

https://www.geeksforgeeks.org/correlating-branch-prediction/

branch target buffer - https://www.youtube.com/watch?v=P16btB7X2mc&ab_channel=PadraicEdgington

tournament branch predictor - https://www.youtube.com/watch?v=-T8yDJ8vTuI&list=PLeWkeA7esB-PcOTrTCvAsaCArnCMQkcNv&index=10&ab_channel=Prof.Dr.BenH.Juurlink

Branch history table -

https://www.youtube.com/watch?v=yk-U6qqeGE0&ab_channel=GATEBOOKVIDEOLECTURES

▼ Tomasulo Algo

https://www.youtube.com/watch?v=y-N0Dsc9LmU&ab_channel=Prof.Dr.BenH.Juurlink

https://www.youtube.com/watch?v=YH2fFu-35L8&ab_channel=Prof.Dr.BenH.Juurlink

https://www.youtube.com/watch?v=vJeqr9KXqqs&ab_channel=AllAboutCSIT

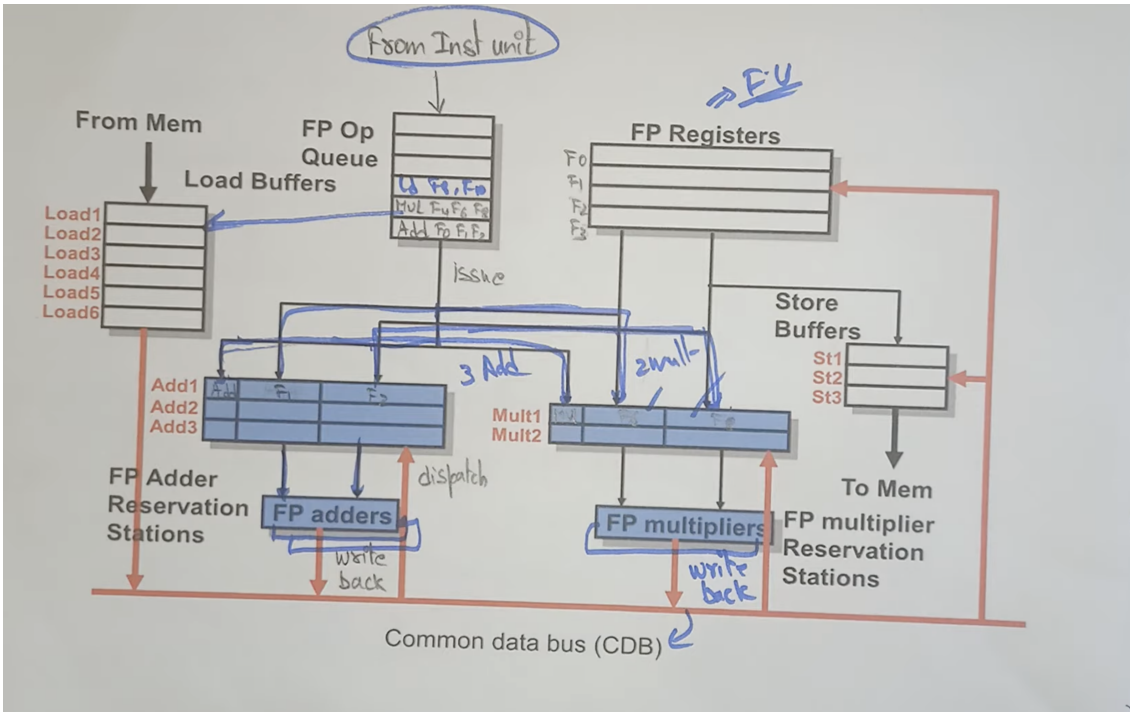
Tomasulo's Algorithm

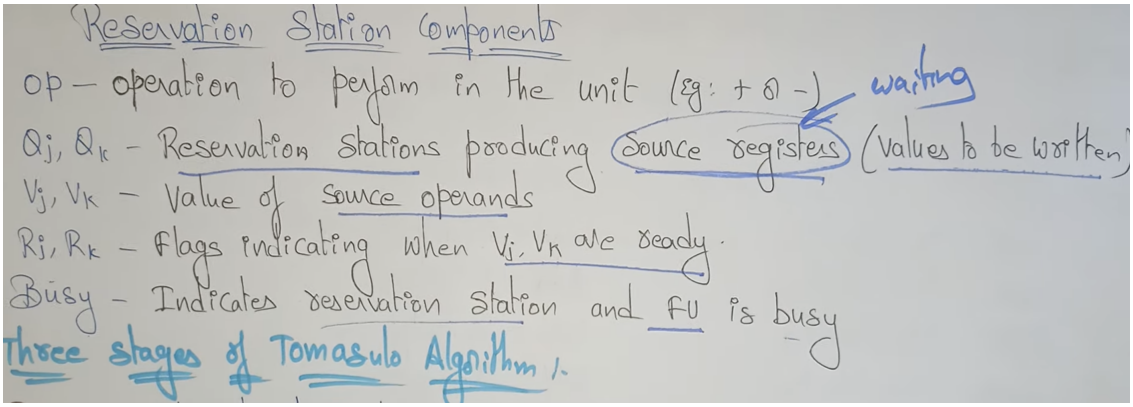
→ It is a method of implementing dynamic scheduling.

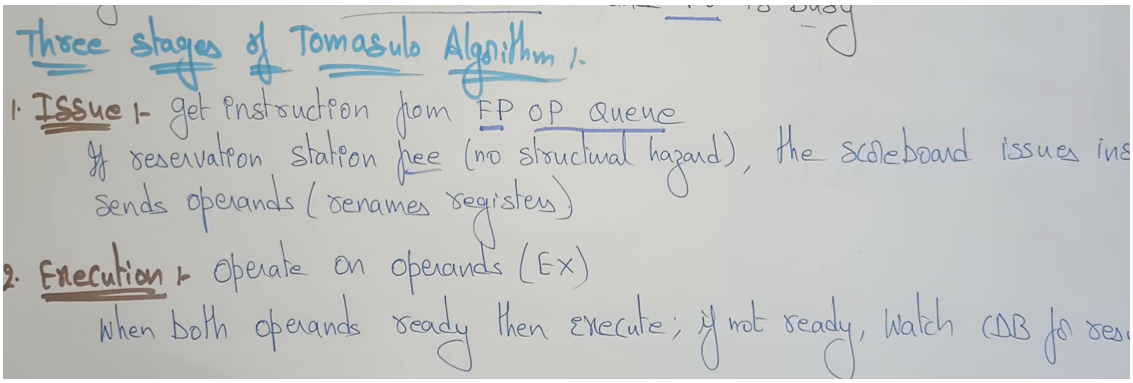
→ It is invented by Robert Tomasulo

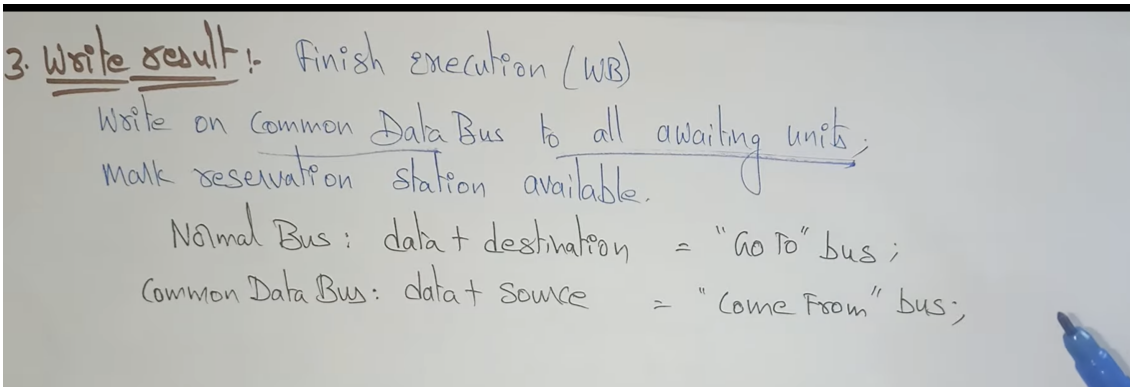
→ It remove name dependency through register renaming

→ It tracks when operands are avl to satisfy the data dependency.









▼ Bernstein Condition

https://www.youtube.com/watch?v=kCQrFCVgPKs&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=ZC9jMg4-8x0&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=20&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=-t0fkuGQICc&list=PLGuh2K9TUN4Qxkrylyq_qwfhAZiuzZ2gz&index=26&ab_channel=AllAboutCSIT

Playlists -

https://www.youtube.com/playlist?list=PLGuh2K9TUN4Qxkrylyq_qwfhAZiuzZ2gz

Notes -

http://www.mmmut.ac.in/News_content/10534tpnews_11082020.pdf

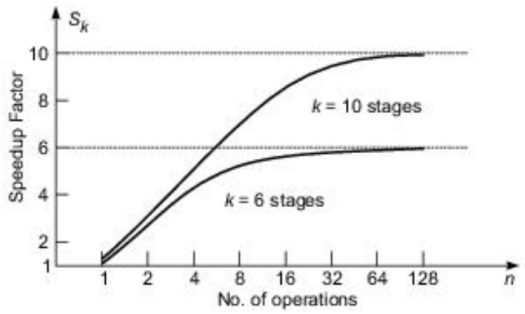
▼ sessional answers -

optimal number of stages -

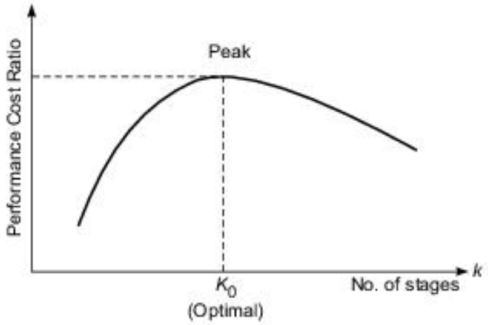
Optimal Number of Stages In practice, most pipelining is staged at the functional level with $2 \le k \le 15$. Very few pipelines are designed to exceed 10 stages in real computers. The optimal choice of the number of pipeline stages should be able to maximize the performance/cost ratio for the target processing load.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $p = t/k + d$, where d is the latch delay. Thus, the pipeline has a maximum throughput of $f = 1/p = 1/(t/k + d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{c + kh} = \frac{1}{(t/k + d)(c + kh)} \tag{6.6}$$



(a) Speedup factor as a function of the number of operations (Eq. 6.5)



(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Fig. 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit

Figure 6.2b plots the PCR as a function of k. The peak of the PCR curve corresponds to an optimal choice for the number of desired pipeline stages:

$$k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}} \tag{6.7}$$

where t is the total flow-through delay of the pipeline. Thus the total stage cost c, the latch delay d, and the latch cost h must be considered to achieve the optimal value k_0.

2021 ans 2 -

1. To compute the speedup obtained from the fast mode we must work out the execution time without the enhancement. We know that the accelerated execution time consisted of two halves: the unaccelerated phase (50%) and the accelerated phase (50%).

   Without the enhancement, the unaccelerated phase would have taken just as long (50%), but the accelerated phase would take 10 times as long, i.e. 500%. So the relative execution time without the enhancement would be $50\% + 500\% = 550\%$.

   Thus the overall speedup is

$$\frac{\mathrm{executiontime}_{\mathrm{unaccelerated}}}{\mathrm{executiontime}_{\mathrm{accelerated}}} = \frac{550\%}{100\%} = 5.5$$

2. To find the percentage of the original execution time which was accelerated, we plug these figures into Amdahl's Law again:

$$\begin{aligned} \mathrm{fractionvectorised} &= \frac{\mathrm{speedup}_{\mathrm{overall}} \times \mathrm{speedup}_{\mathrm{accelerated}} - \mathrm{speedup}_{\mathrm{accelerated}}}{\mathrm{speedup}_{\mathrm{overall}} \times \mathrm{speedup}_{\mathrm{accelerated}} - \mathrm{speedup}_{\mathrm{overall}}} \\ &= \frac{5.5 \times 10 - 10}{5.5 \times 10 - 5.5} \\ &= 45/49.5 = 90.90\% \end{aligned}$$

2021 ans 3-

https://www.slideshare.net/mahinthjoe/ch6pr - page 23

- amdhals law
- non linear pipeline
- reservation table
- performance
- arithmetic mean
- harmonic mean

principle of scalable performace -

https://www.youtube.com/watch?v=eVIuzh5VMVA&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=8&ab_channel=ShanuKuttanCSEClasses

system performance -

https://www.youtube.com/watch?v=N5BmtcdtMVw&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=12&t=429s&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=Xxj9PND9iwQ&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=13&ab_channel=ShanuKuttanCSEClasses

non linear pipleline -

https://www.youtube.com/watch?v=O1S9JxWitaM&list=PL3R9-um41JsxFEluRokmVwMp3zF7JL7tL&index=7&t=1034s&ab_channel=ShanuKuttanCSEClasses

reservation table -

https://www.youtube.com/watch?v=2PIj3YekQlk&list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6&index=31&ab_channel=ShanuKuttanCSEClasses

amdhal law -

https://www.youtube.com/watch?v=yi3zHpgGXpk&t=55s&ab_channel=ShanuKuttanCSEClasses

amdhal and gustafson law -

https://www.slideshare.net/anshuljmi/parallel-programming-20529860

https://www.youtube.com/watch?v=mZau0RI1Wvs&t=151s&ab_channel=shivaashirvaad

notes-

http://www.mmmut.ac.in/News_content/10534tpnews_11082020.pdf

playlist -

https://www.youtube.com/playlist?list=PL3R9-um41Jsz4as9nqgVB6YRR90rs0wE6

pyq -

https://www.slideshare.net/mahinthjoe/ch6pr — pg 24

Solution

$$\begin{aligned}
\frac{Exectime_{new}}{Exectime_{old}} &= (1 - F_{enhanced}) + F_{enhanced} \times \frac{1}{speedup_{enhanced}} \\
&= (1 - F_{enhanced}) + 0.1 \times F_{enhanced} \\
&= 1 - 0.9 \times F_{enhanced} \\
\frac{Exectime_{old}}{Exectime_{new}} &= (1 - F_{dehanced}) + F_{dehanced} \times \frac{1}{speedup_{dehanced}} \\
&= 0.5 + \frac{0.5}{0.1} \\
&= 5.5 \\
5.5 &= \frac{1}{1 - 0.9 \times F_{enhanced}} \\
5.5 - 4.95 \times F_{enhanced} &= 1 \\
F_{enhanced} &= \frac{4.5}{4.95} \\
&= 0.91
\end{aligned}$$

9/13

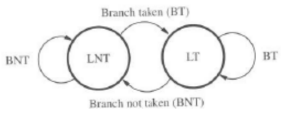
Check

$$
\begin{aligned}
Speedup &= \frac{1}{(1-0.91)+0.91\times\frac{1}{10}} \\
&= \frac{1}{0.09+0.091} \\
&= \frac{1}{0.181} \\
&= 5.52
\end{aligned}
$$

▼ Branch Prediction -

http://web.cecs.pdx.edu/~zeshan/ece586_lec17_1.pdf

1-bit Branch Prediction

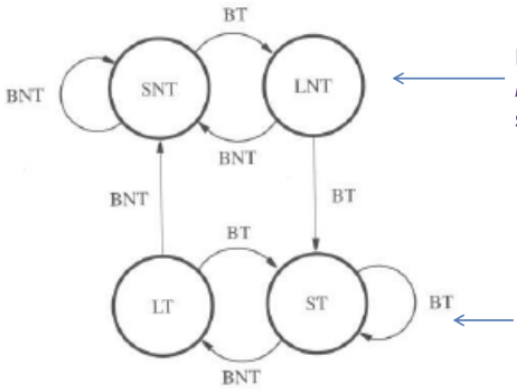


- The algorithm is implemented by a 2-state state machine:
  - LT -- Branch is likely to be taken
  - LNT -- Branch is likely not to be taken
- The prediction for a branch is based on the *current state* of the state machine
- The state transitions are based on the actual outcome computed after the branch has been executed

Example

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

2-bit Branch Prediction



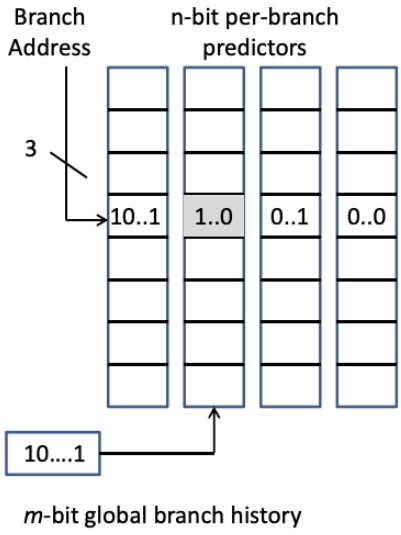
ST: Strongly likely to be taken
LT: Likely to be taken
LNT: Likely not to be taken
SNT: Strongly likely not to be taken

Correlating Branch Predictor with m-bit Global History Register

(m,n) correlating predictor uses behavior of last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor

Total number of bits = 2^m * n * Number of entries in each prediction table
= 2^m * n * $2^{(\text{branch address bits used})}$

For a predictor that does not use any global history, m = 0, e.g., a (0,2) is a 2-bit predictor with no global history



Correlating Predictor Examples

Question: How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Solution:

Number of bits = $2^m * n * 2^{(\text{branch address bits used})}$

For the (0,2) predictor:

Number of bits = $2^0 * 2 * 4K = 8K$ bits

For the (2,2) predictor:

Number of bits = 8K

$8K = 2^2 * 2 *$ Number of predictor entries

=> Number of predictor entries = 1K

18BCS032

(5) Discuss (m,n) branch predictor

The (m,n) branch predictor is also known as correlated branch prediction. It is a two level branch prediction in which prediction accuracy is improved as it takes into consideration the recent behaviour of other branches also.

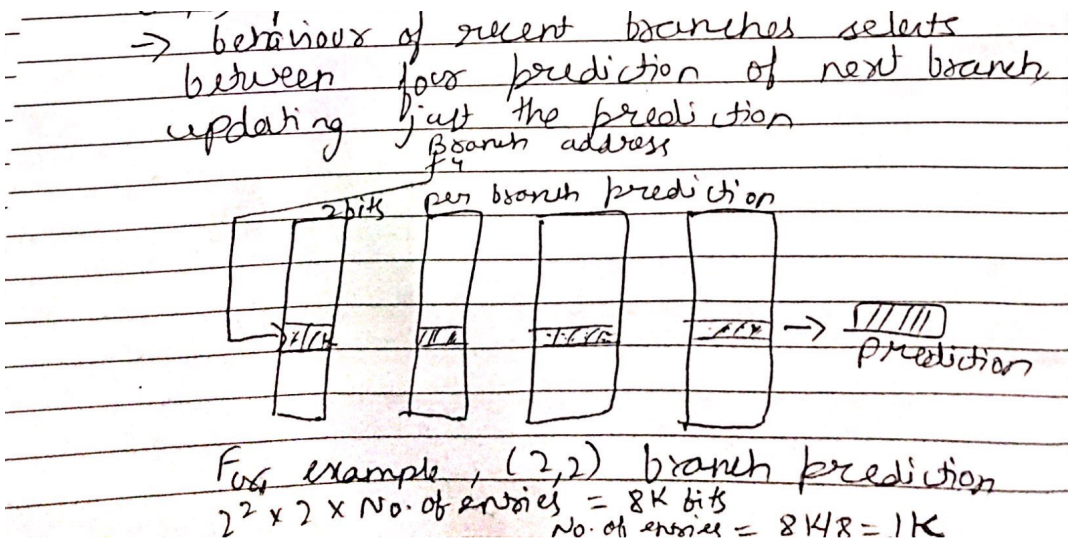
1. It uses k least significant bits of branch target address which is fetched before

2. It also uses local history table (LHT) which is table of shift registers where shift registers refers to the test outcome of m branches having store k least significant bits.

3 It also uses local prediction table to predict the outcome depending on the state in which it is present

(2,2) predictor

→ behaviour of recent branches selects between four prediction of next branch updating just the prediction



For example, (2,2) branch prediction

$2^2 \times 2 \times$ No. of entries = 8K bits

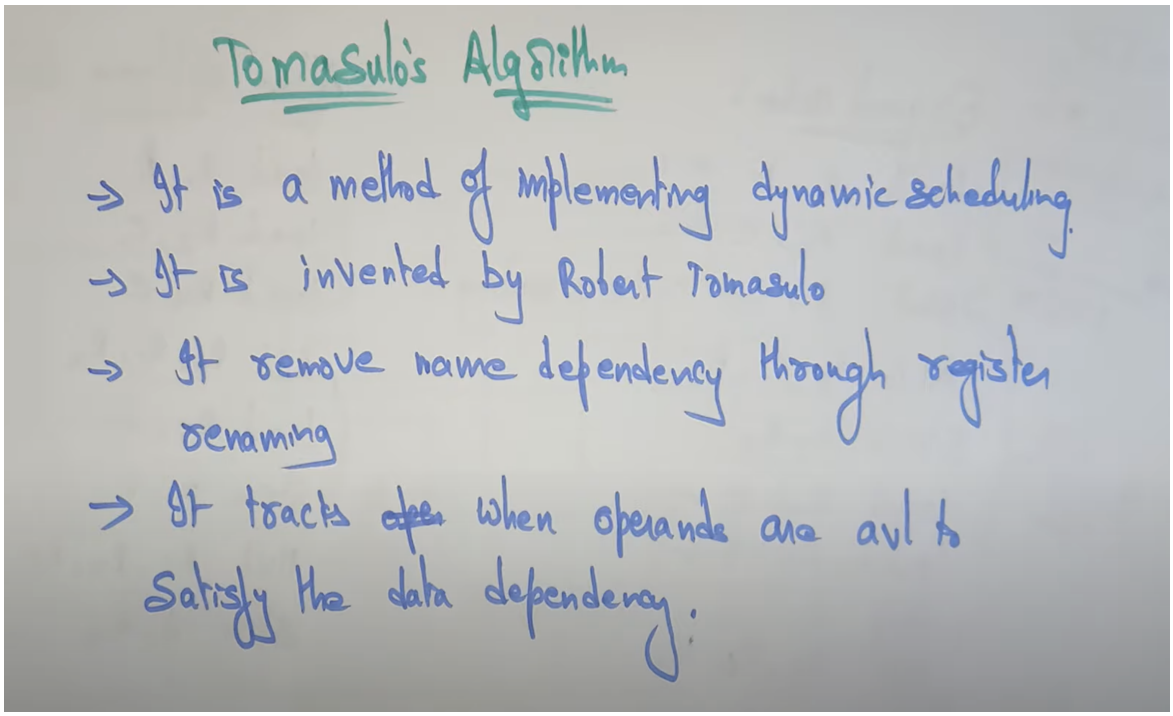
No. of entries = 8K/8 = 1K

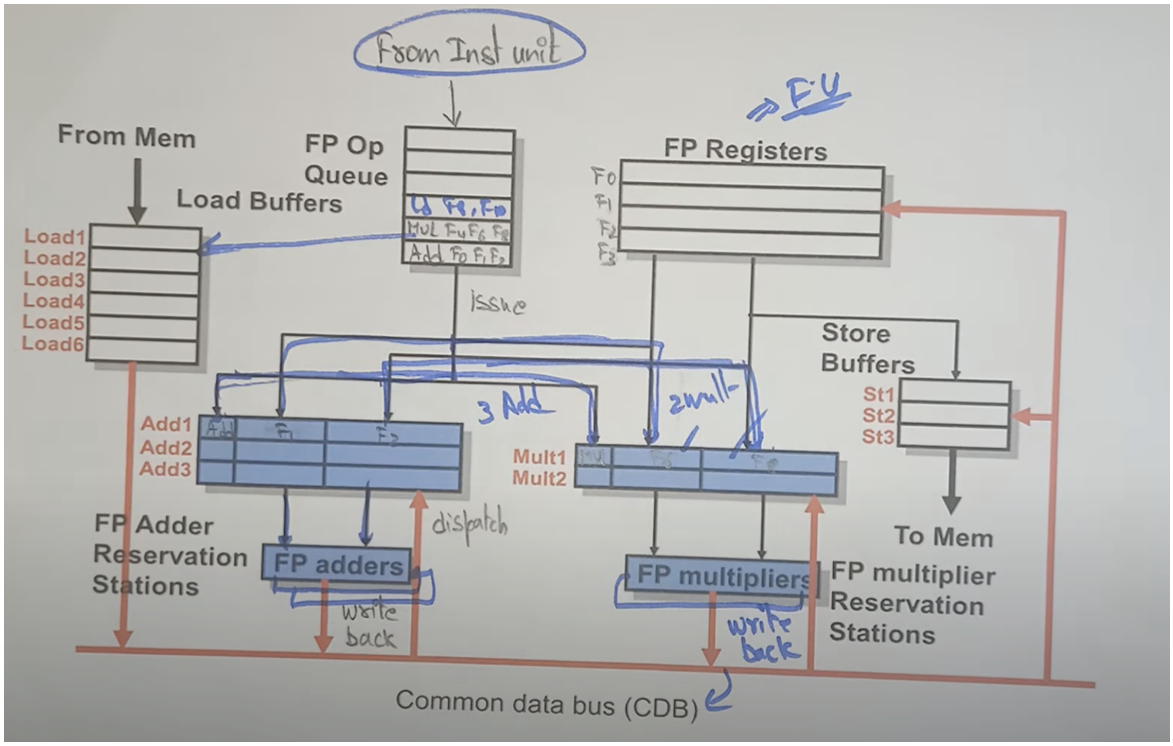
Tournament predictors: The next type of predictor is a tournament predictor. This uses the concept of —Predicting the predictorǁ and hopes to select the right predictor for the right branch. There are two different predictors maintained, one based on global information and one based on local information, and the option of the predictor is based on a selection strategy. For example, the local predictor can be used and every time it commits a mistake, the prediction can be changed to the global predictor. Otherwise, the switch can be made only when there are two successive mispredictions. Such predictors are very popular and since 2006, tournament predictors using » 30K bits are used in processors like the Power5

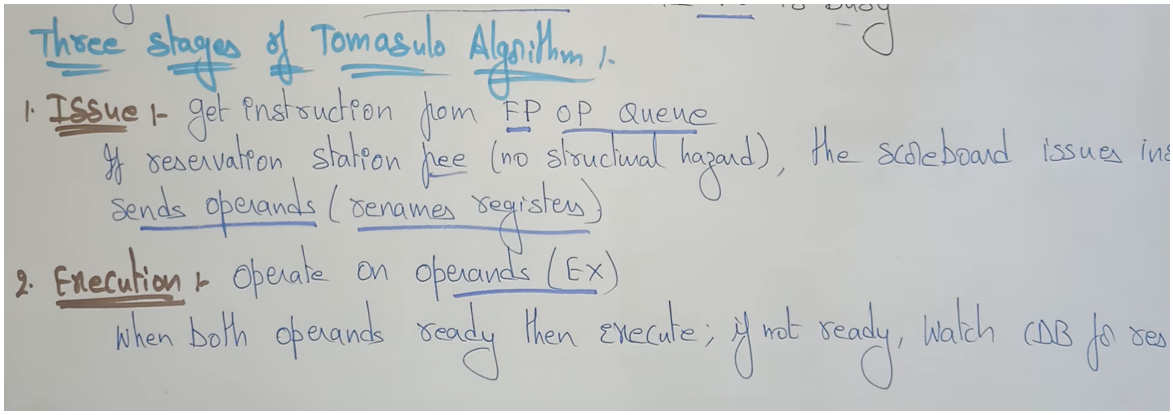
and Pentium 4. They are able to achieve better accuracy at medium sizes (8K – 32K bits) and also make use of very large number of prediction bits very effectively. They are the most popular form of *multilevel branch predictors* which use several levels of branch-prediction tables together with an algorithm for choosing among the multiple predictors.

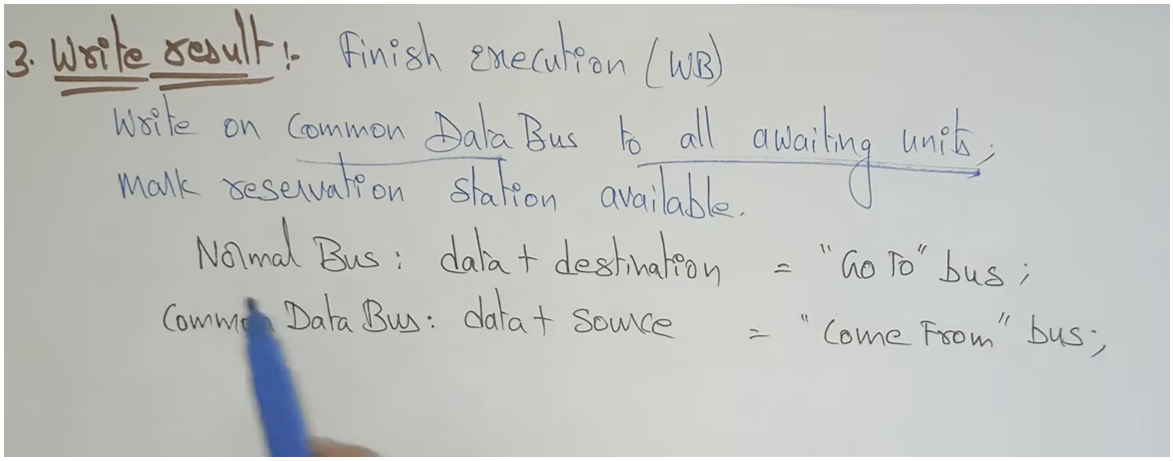
▼ Tomusolu Algorithm

https://www.youtube.com/watch?v=vJeqr9KXqqs&ab_channel=AllAboutCSIT









▼ Amdhal law

Amdahl's Law[1]

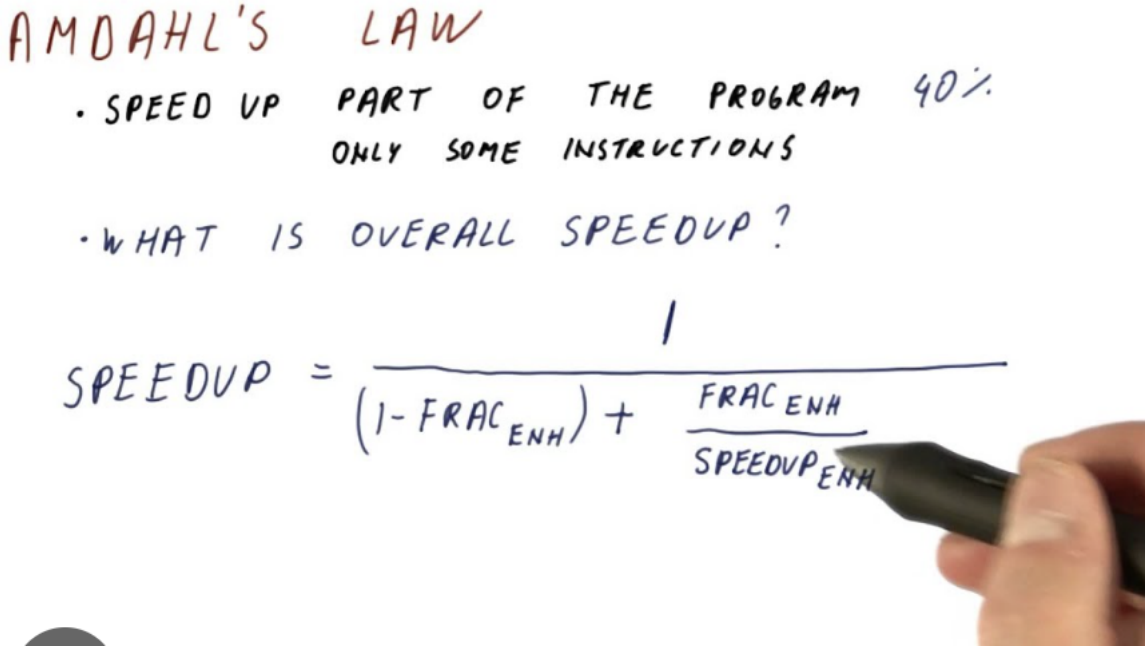
Amdahl's law is used to find the maximum expected improvement to an overall system when only part of the system is parallelized. That is, the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains sequential), then the maximum speedup that can be achieved by using N processors is:

$$SpeedUp(P, N) = \frac{1}{(1-P) + \frac{P}{N}}$$

As N tends to infinity, the maximum speedup tends to $1 / (1 - P)$.

658 × 305



▼ Reservation table done

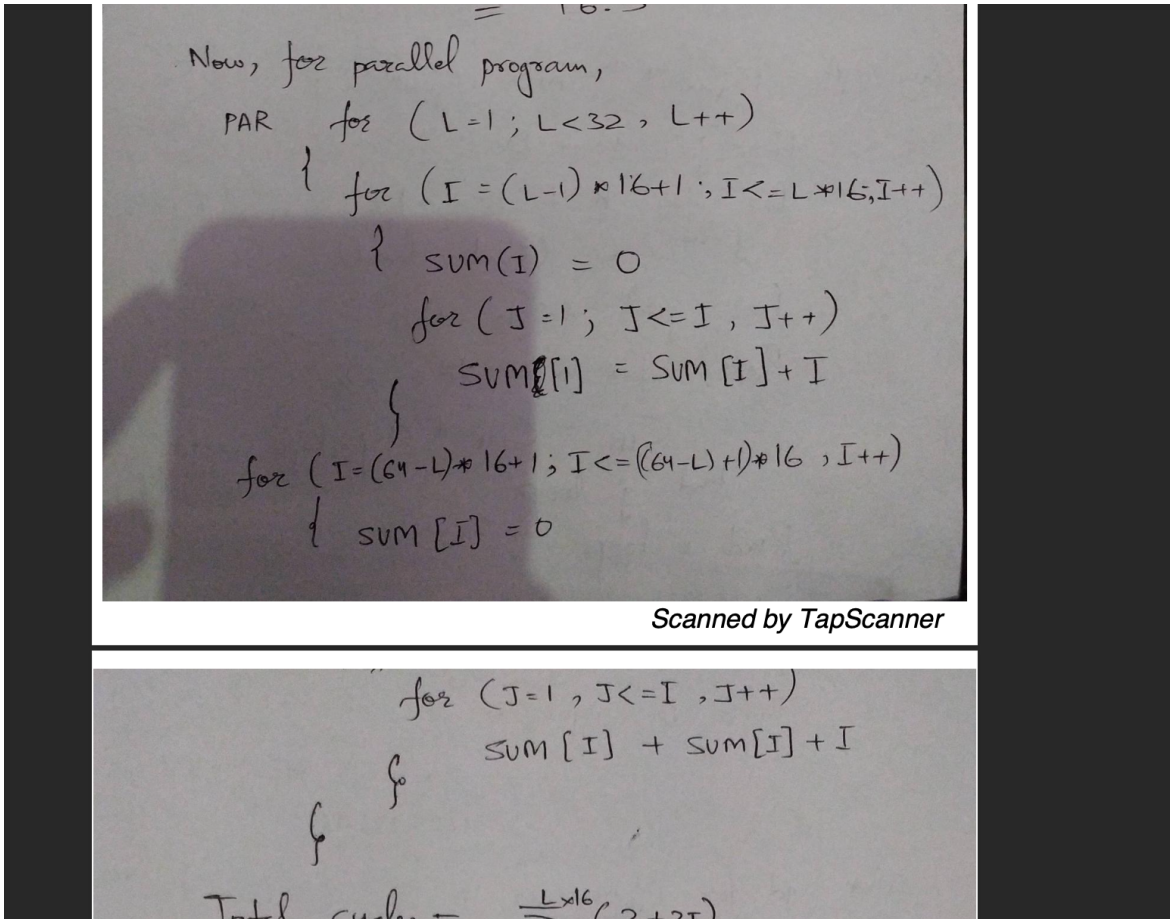
https://www.youtube.com/watch?v=7AdfBuh9AdE&list=PL3R9-um41JswxXxLu6qmsib45wDd7d0YU&index=30&ab_channel=ShanuKuttanCSEClasses

https://www.youtube.com/watch?v=ITdw2mGQWIk&list=PL3R9-um41JswxXxLu6qmsib45wDd7d0YU&index=31&ab_channel=ShanuKuttanCSEClasses

▼ Distributed System Paradigms

https://www.youtube.com/watch?v=jQK-iji0ORU&ab_channel=CSEDepartmentEngineeringCollege,Jhalawar

Question -



ans i) total no. of cycles as a uniprocessor system:

$$= \sum_{i=1}^{4096} (2 + 2I)$$

$$= 2 \times 4096 + 4096 \times 4097$$

$$= 8192 + 16781312$$

$$= 16789504$$ Ans

ii) in a multiprocessor system

(ii) in a multiprocessor system

time taken by first processor $= \sum_{i=1}^{64} (2+2I)$

$= 4288$

time taken by second processor $= \sum_{i=65}^{128} (2+2I)$

$= 12480$

time taken by 64th processor $= \sum_{i=4033}^{4096} (2+2I)$

$= 520384$

Speedup time $= 16789504 / 520384$

$= \underline{32.26}$ Ans

```
    for ( J = 1 ; J <= I ; J++ )
        sum [I] = sum [I] + I
    }
for ( I = (128 - L) * 32 + 1 ; I <= (128 - L + 1) * 32, I++) {
    sum [I] = 0
    for ( J = 1 ; J <= I ; J++ )
        sum [I] = sum [I] + I
}
```

iv) total no. of cycles =

$$\sum_{i=(L-1)*32+1}^{L*32} (2+2I) + \sum_{I=(128-L)\times 32+1}^{(128-L+1)\times 32} (2+2I)$$

$$= 262336$$

Now,

Speed up = 16789504 / 262336

Q. 3 a) SIMD algorithm for N×N matrix multiplication

```
begin
  {stagger matrices}
  for k := 1 to n-1 do
    for all P(i,j) do
      if i > k then
        A(i,j) ← A(i, j+1)
      endif
      if j > k then
        B(i,j) ← B(i+1, j)
      endif
    endfor-all
  endfor
  {compute dot products}
  for-all P(i,j) do
    C(i,j) := A(i,j) × B(i,j)
  endfor-all
  for k := 1 to n-1 do
    for-all P(i,j)
      A(i,j) ← A(i, j+1)
      B(i,j) ← B(i+1, j)
      C(i,j) := C(i,j) + A(i,j) × B(i,j)
    endfor-all
  endfor
end
```

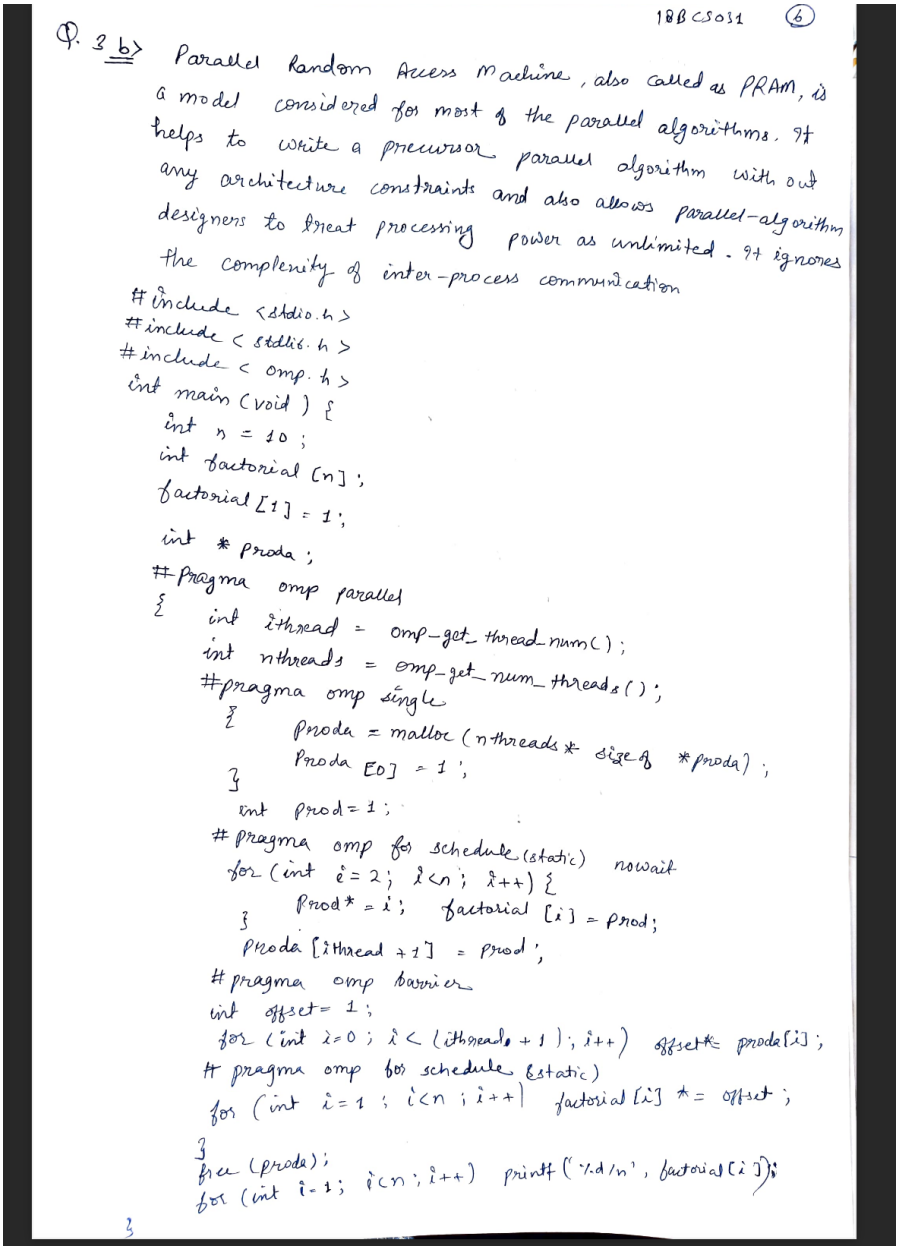
Qno2) Ans Required SIMD algorithm :

For j=1 to n Do
Par for k=1 to n Do
$I^{Cik} = 0$ (vector load)
for j=1 to n Do
Par for k=1 to n Do
$C_{ik} = C_{ik} + a_{ij} \cdot b_{jk}$ (vector multiply)
End j loop
End i loop

C_{n1} C_{n2} C_{nn}

for i=1 to n do
for k=1 to n do in parallel
$c_{i,k} = 0$
end for
for j=1 to n do
for k=1 to n do in parallel
$C_{i,k} = C_{i,k} + a_{i,j} * b_{j,k}$
end for
end for
end for

Time Complexity: $O(n^2)$.



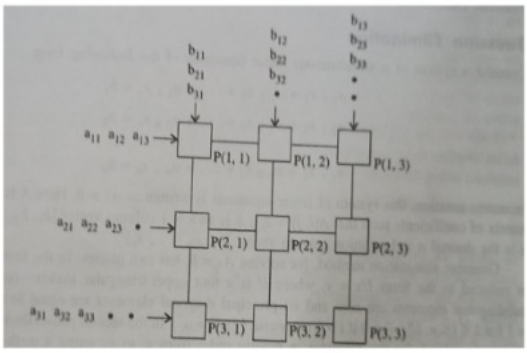
18BCS031 (6)

Q. 3 b) Parallel Random Access Machine, also called as PRAM, is a model considered for most of the parallel algorithms. It helps to write a precursor parallel algorithm without any architecture constraints and also allows parallel-algorithm designers to treat processing power as unlimited. It ignores the complexity of inter-process communication

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void) {
  int n = 10;
  int factorial[n];
  factorial[1] = 1;
  int *proda;
  #pragma omp parallel
  {
    int ithread = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    #pragma omp single
    {
      proda = malloc(nthreads * sizeof *proda);
      proda[0] = 1;
    }
    int prod = 1;
    #pragma omp for schedule(static) nowait
    for (int i = 2; i<n; i++) {
      prod *= i; factorial[i] = prod;
    }
    proda[ithread+1] = prod;
    #pragma omp barrier
    int offset = 1;
    for (int i=0; i<(ithread+1); i++) offset *= proda[i];
    #pragma omp for schedule(static)
    for (int i=1; i<n; i++) factorial[i] *= offset;
  }
  free(proda);
  for (int i=1; i<n; i++) printf("%d\n", factorial[i]);
}
```

- pram matrix multiplication

Matrix Multiplication on Mesh with NxN processors



Parallel Algorithm

```
for i := 1 to n do in parallel
    for j := 1 to n do in parallel
        c_{i,j} := 0
        while P_{i,j} receives two inputs a and b do
            c_{i,j} := c_{i,j} + a * b
            if i < n then send b to P_{i+1, j}
            end if
            if j < n then send a to P_{i, j+1}
            end if
        end while
    end for
end for
```

Complexity

- Processor P(I,j) receives its input after i-1+j-1 steps from the beginning of computation
- After getting the input P(I,j) takes n steps to Compute C(I,j)
- So C(I,j) is computed in i-1+j-1 +n steps
- So complexity of algorithm is O(i-1+j-1 +n)=O(n-1+n-1+n)=O(n)

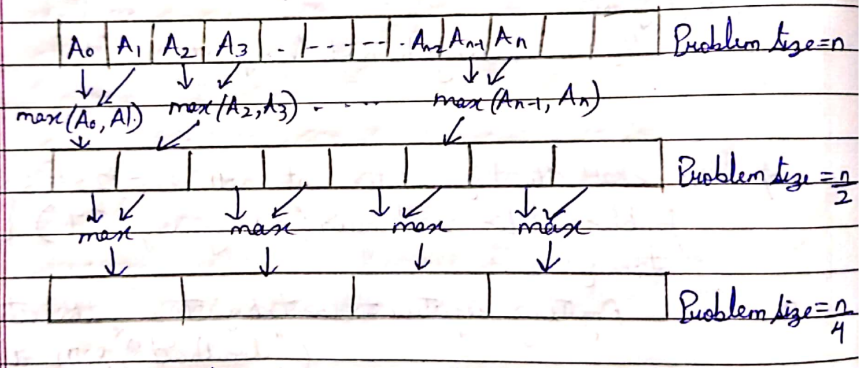
- pram for max of numbers -
-

Q-3. | b) PRAM (Parallel Random Access Machine)

→ abstract machine for designing the algorithms applicable to parallel computers.

→ M' is a system $<M, X, Y, A>$ of infinitely many RAMs $M_1, M_2 \ldots$ each M_i is called a processor of M'. All the processors are assumed to be identical. Each has ability to recognize its own index i.

- Input cells $X(1), X(2) \ldots$
- Output cells $Y(1), Y(2) \ldots$
- Shared memory cells $A(1), A(2) \ldots$



Since the problem size is decreasing in half every

Since the problem size is decreasing in half every clock cycle. The time taken is determined by height of the above process = log n.
Hence time complexity = O(log n)

Algo:
Program in P(i)
L = n
Repeat

Scanned with CamScanner

Shubham Gupta
17BCS017

DATE: __/__/____
PAGE _____

```
if (i<L) then begin
    read A[i] from shared memory
    Read A[i+1] from shared memory
    Compute max(A[i], A[i+1])
    Store in A[i][2]
    i = i+2
L = L/2
Do until (L=1)
```

- open mp pi program

https://www.youtube.com/watch?v=s6CE8avVqeo&ab_channel=NitinRanjan18BCE0272

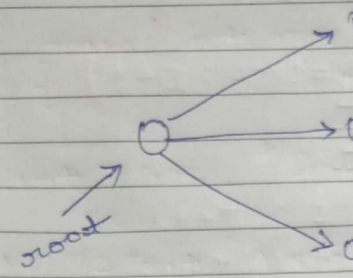
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8

static long steps = 1000000000;
double step;

int main (int argc, const char *argv[]) {

    int i,j;
    double x;
    double pi, sum = 0.0;
    double start, delta;

    step = 1.0/(double) steps;

    // Compute parallel compute times for 1-MAX_THREADS
    for (j=1; j<= MAX_THREADS; j++) {

        printf(" running on %d threads: ", j);

        // This is the beginning of a single PI computation
        omp_set_num_threads(j);

        sum = 0.0;
        double start = omp_get_wtime();


        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0; i < steps; i++) {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }

        // Out of the parallel region, finialize computation
        pi = step * sum;
        delta = omp_get_wtime() - start;
        printf("PI = %.16g computed in %.4g seconds\n", pi, delta);

    }


}
```

based on pi = 4 * area of curcle/ area of square, circle is inside square and no of points equi to area

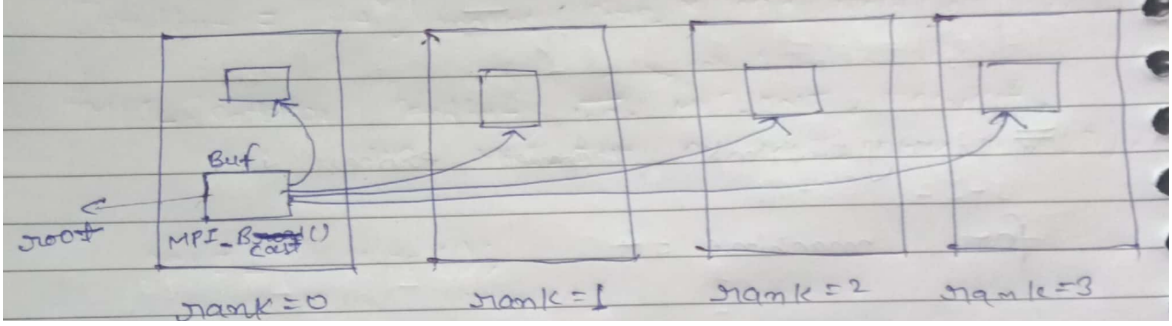
-

☆ MPI Broadcast :-

Date .....



Syntax :




```
MPI_Bcast (*buf, count, datatype, root, communicator)
```

→ root : who is broadcasting.

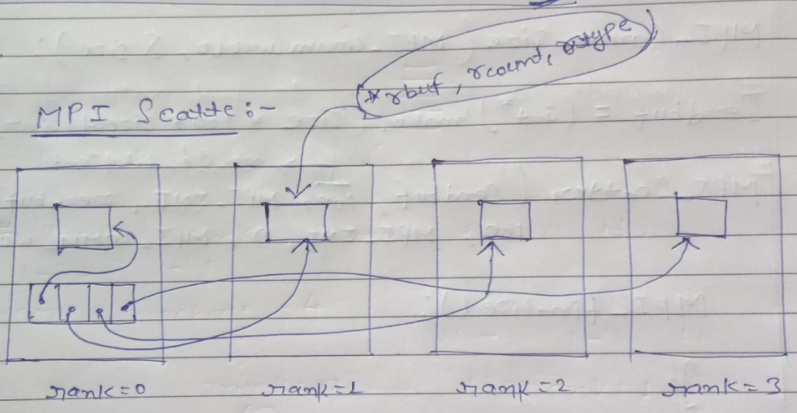
ex matrix A[ ]$_{m \times n}$ Send to all ?

```
double A[N][N];
MPI_BCast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
double A[N][N];
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
{
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            A[i][j] = i+j;
}
```

Date .....

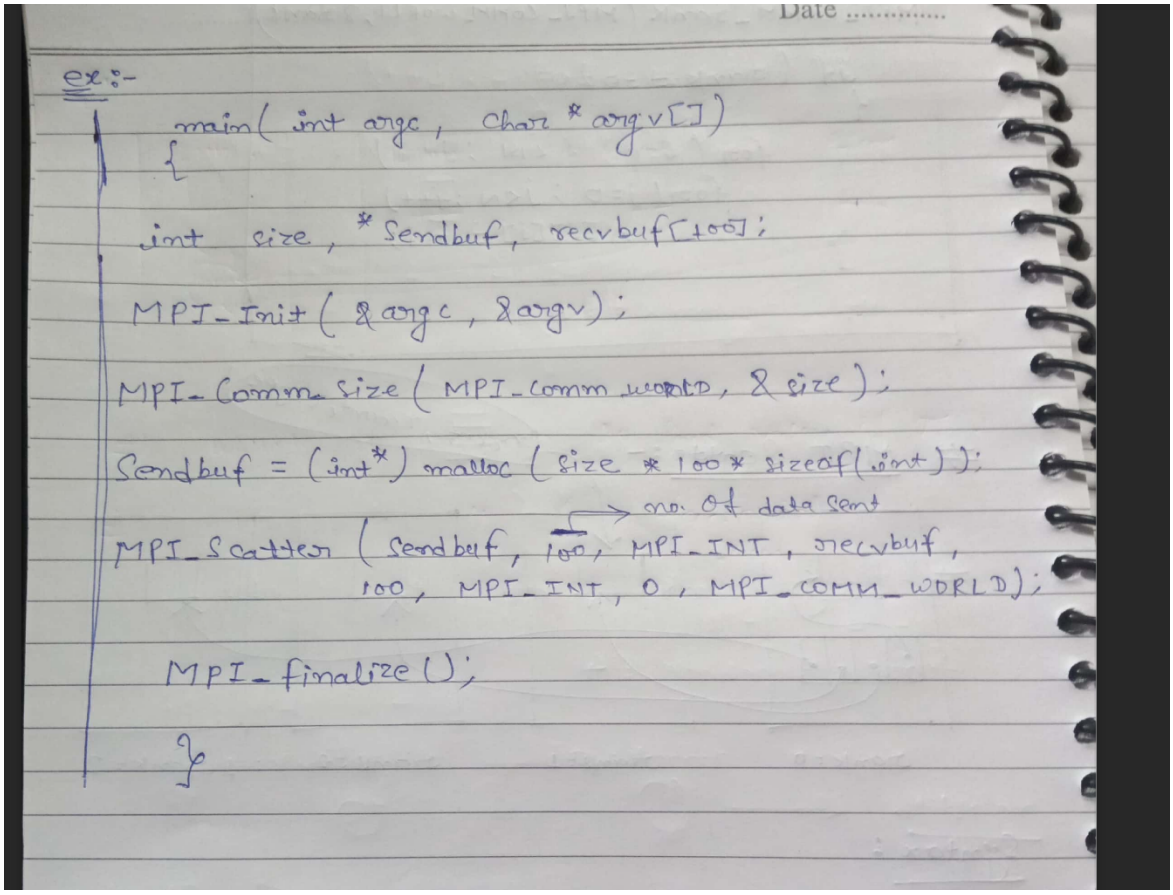
MPI Scatter:-



Syntax:

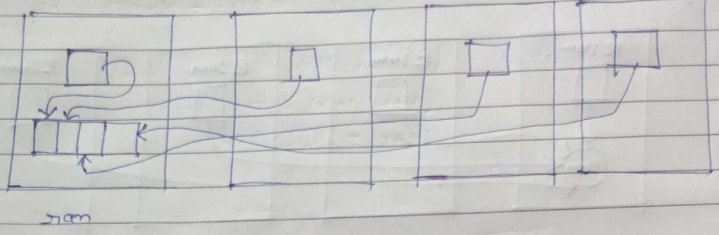
MPI_Scatter(*Sbuf, Scount, Stype, *rbuf, rcount, rtype, root, communicator)

ex


Date .....
ex:-
main(int argc, char *argv[])
{
int size, *Sendbuf, recvbuf[100];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_Comm_world, &size);
Sendbuf = (int*) malloc(size * 100 * sizeof(int));
no. of data sent
MPI_Scatter (Sendbuf, 100, MPI_INT, recvbuf,
100, MPI_INT, 0, MPI_COMM_WORLD);
MPI_finalize();
}

Gather :- → opposite of Scatter

Date .....



ram

Syntax :-

MPI_Gather (*Sbuf, Scount, stype, *rbuf, rcount, rtype, root, comm);

for program replace Scatter by Gather.

Reduce :-

Syntax :-

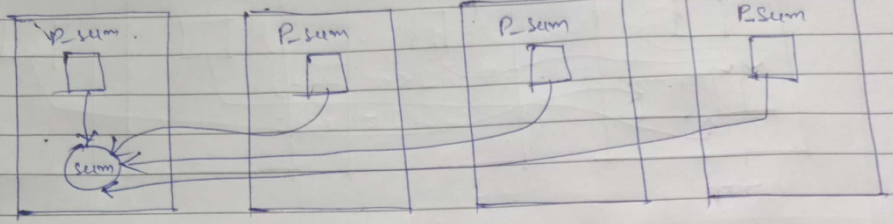
MPI_Reduce (*Sbuf, *recivebuf, count, datatype, op, root, comm);

Op = operation → MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD

MPI.org

Date .....

ex

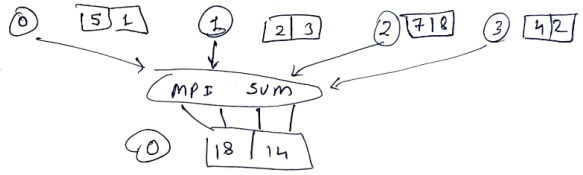


Q. 4a)

i) MPI reduce :-

```
MPI_reduce (
        void * send_data,
        void * recv_data,
        int count,
        MPI_Datatype datatype,
        MPI_Op op,
        int root,
        MPI_Comm communicator)
```

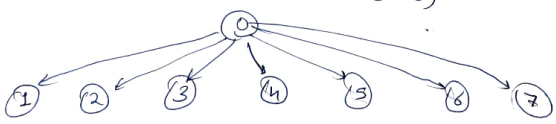
e.g



In above each process contains 2 integers. MPI_reduce is called with a root process of 0 and using MPI_SUM as reduced. The i^{th} elem from each array are summed into i^{th} elem from result array

ii) MPI_broadcast

```
MPI_Bcast (
        void * data,
        int count,
        MPI_Datatype datatype,
        int root,
        MPI_Comm communicator)
```

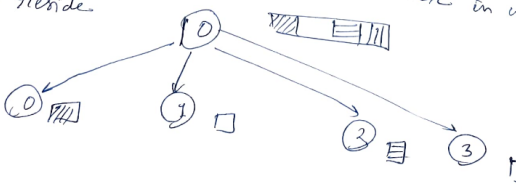


In this process, process 0 is the root process and it has initial copy of data. All other process receives the copy of data.

iii) MPI_Scatter :-

```
MPI_Scatter (
    Void * send_data,
    int send_count,
    MPI_Datatype send_datatype
    Void * recv_data,
    int recv_count,
    MPI_Datatype recv_datatype
    int root,
    MPI_Comm communicator )
```

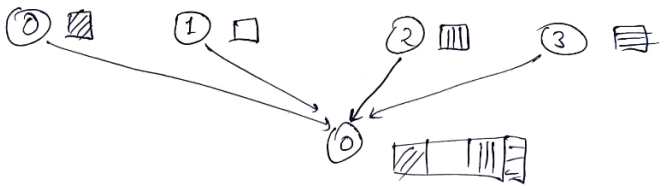
The first parameter send_data is array of data that resides on root process. The second and third parameter send_count and send_datatype dictate how many elements of a specific datatype will go to each process. If send_count is 1 and send_data is MPI_INT, then process zero gets first integer of array, process one gets 2nd integer and so on. The recv_data, recv_count and recv_datatype are similar. The last param. root indicates root that is scattering data and communicator in which processor reside.



iv) MPI_Gather :

Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers these to next process.

```
MPI_Gather (
        void* send_data,
        int send_count,
        MPI_datatype send_datatype,
        void* recv_data,
        int recv_count,
        MPI_Datatype recv_datatype,
        int root,
        MPI_comm communicator)
```



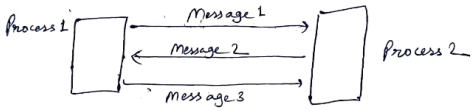
18CS031 (11)

Q. 5 ans There are various types of paradigm for distributed computing:-

⇒ Message Passing Paradigm:- It is a basic approach for inter process communication. The data exchange between the sender and the receiver. A process sends a message representing the request. The receiver receives and processes it, then sends back as reply.

Operations: send, receive

Connections: connect, disconnect

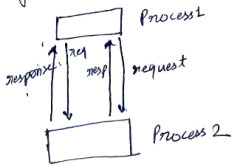


⇒ Client Server Paradigm:- The server acts as a service provider, the client issues the request and wait for response from the server. Here server is dump machine. Until client make a call, server doesn't communicate. Many internet services are client-server applications.

Server Process: listen, accept

Client Process: issue and accept the request

⇒ Peer to peer Paradigm:- Direct communication between processes. Here is no client or server, anyone can make request to others and get the respons example file transfer (P2P).



⇒ Message System Paradigm :- acts as intermediate among independent processes. It also acts as a switch through which process exchange messages asynchronously in decoupled manner. Sender sends message which drop at first in message system then forward to message queue which is associated with receiver.

Types :- → Point to Point message model
→ Publish subscribe model

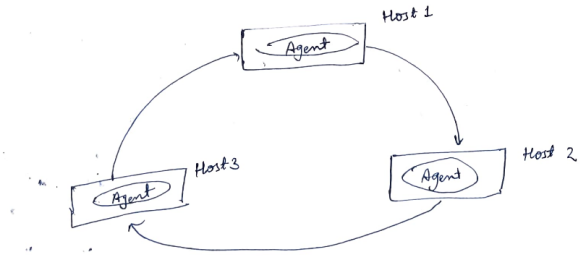
⇒ Distributed object Paradigm :-

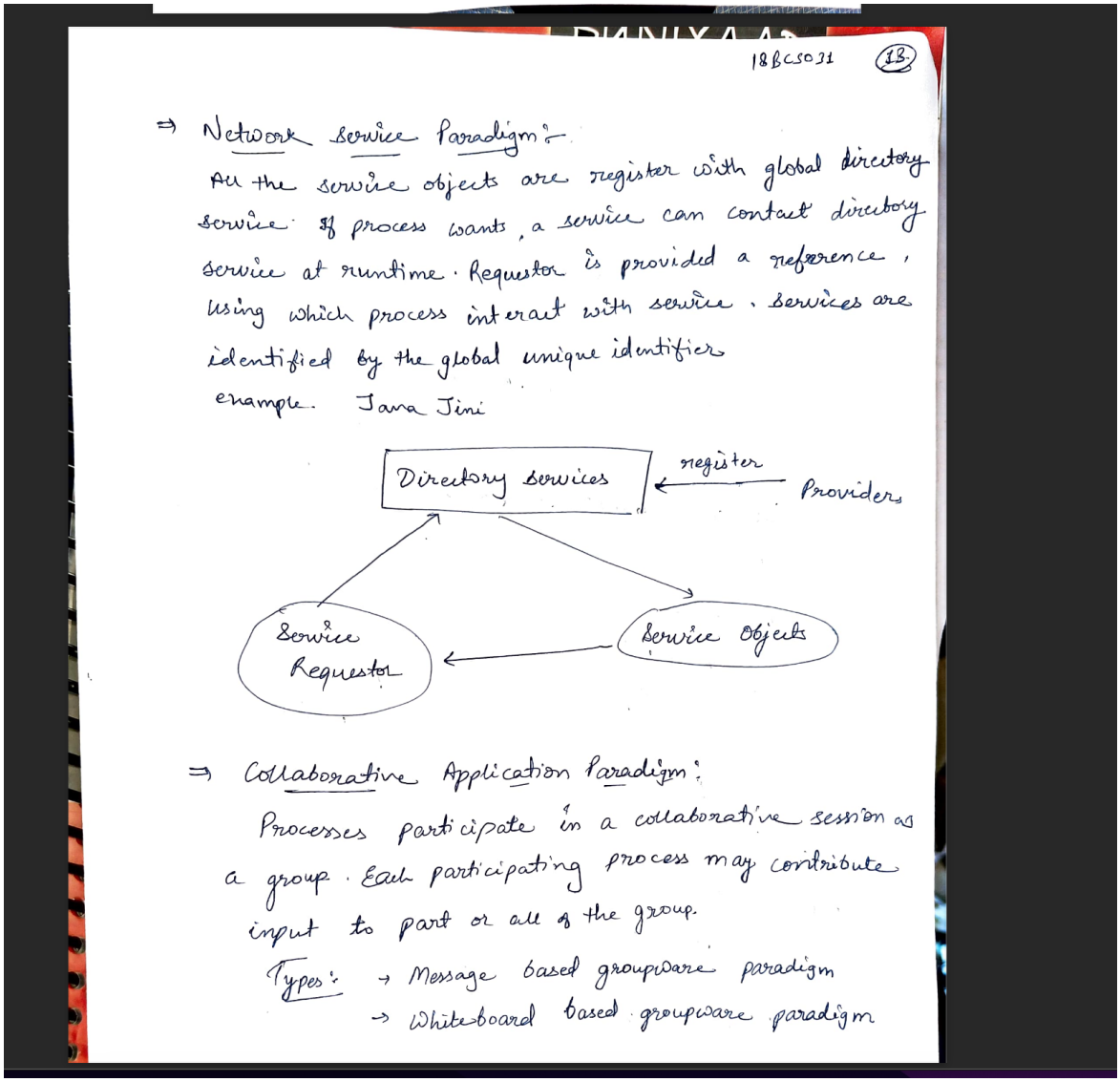
Applications access the objects distributed over the network. Objects provide methods, through the invocation of which an application obtain access to services. Eg. CORBA

Types : → Remote Method Invocation
→ Object Request Broker
→ Object space

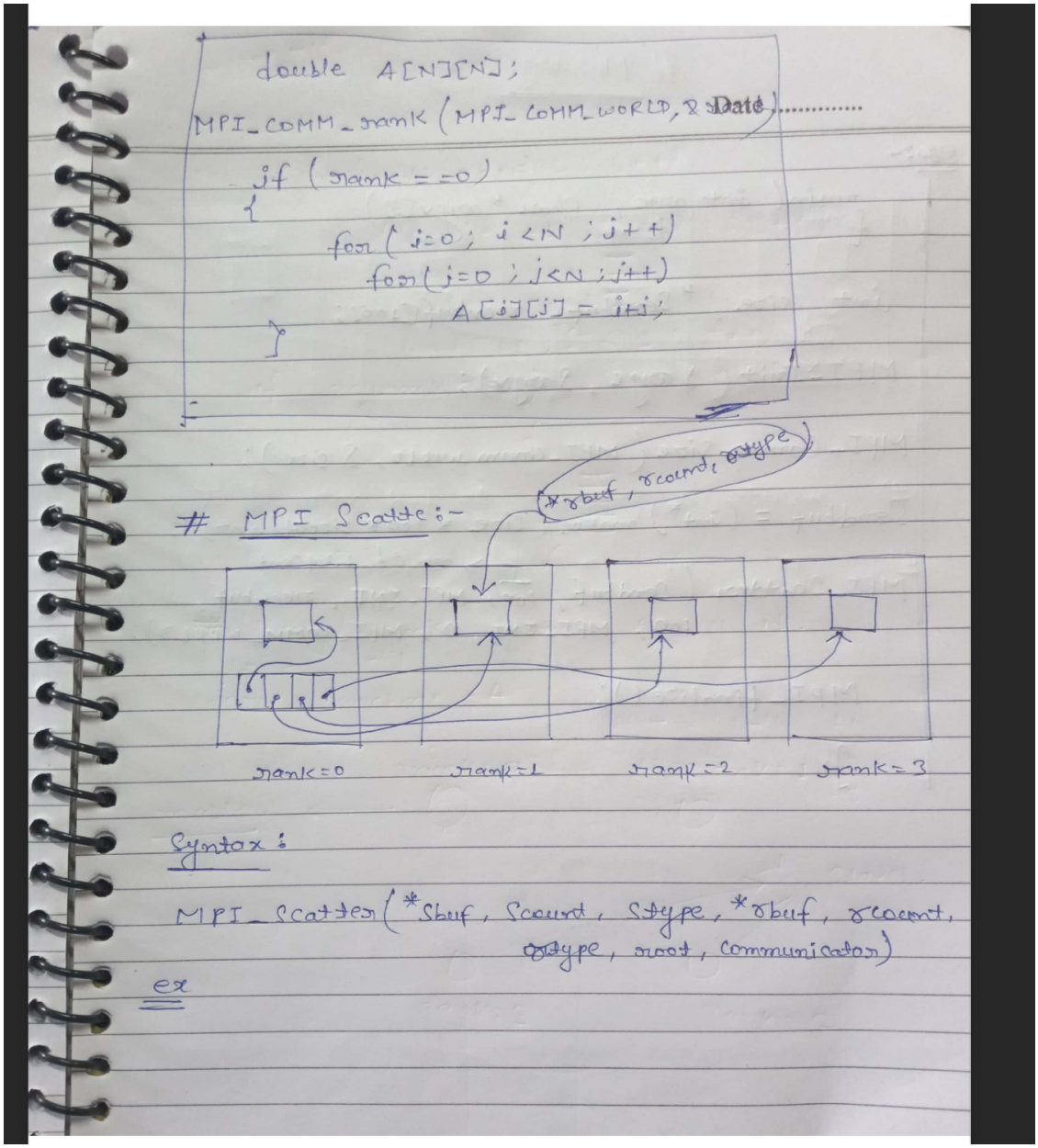
⇒ Mobile Agent Paradigm :-

Mobile Agent starts from originated host and transports over host to host. At each host, the agent can access the services or resources to complete the mission





- ab {a, 1-a} derivation refer copy
- three enhancement question refer copy
- sun ni and amdhal law - Amdahl's law is based on a fixed workload, where the problem size is fixed regardless of the machine size. Gustafson's law is based on a scaled workload, where the problem size is increased with the machine size so that the solution time is the same for sequential and parallel executions. Sun and Ni's law is also applied to scaled problems, where the problem size is increased to the maximum inerory capacity.

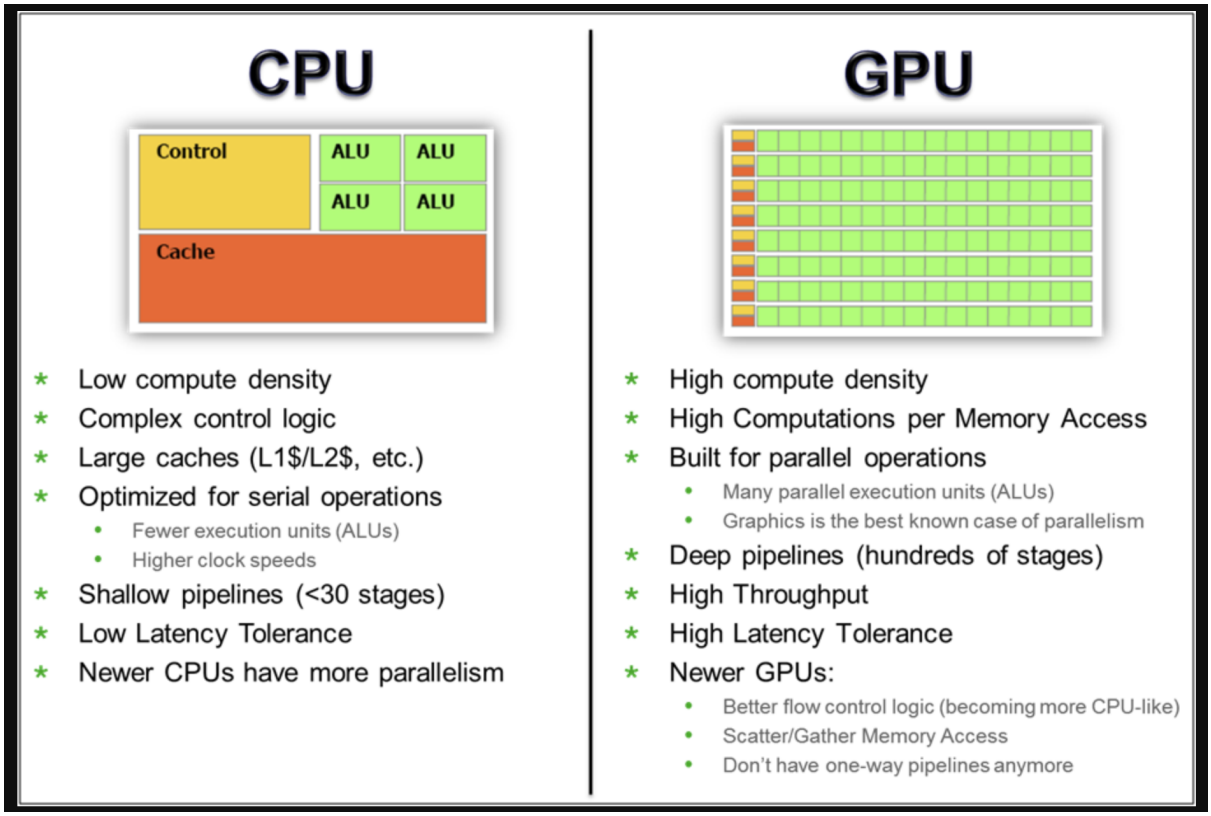


Parallel quick sort analysis

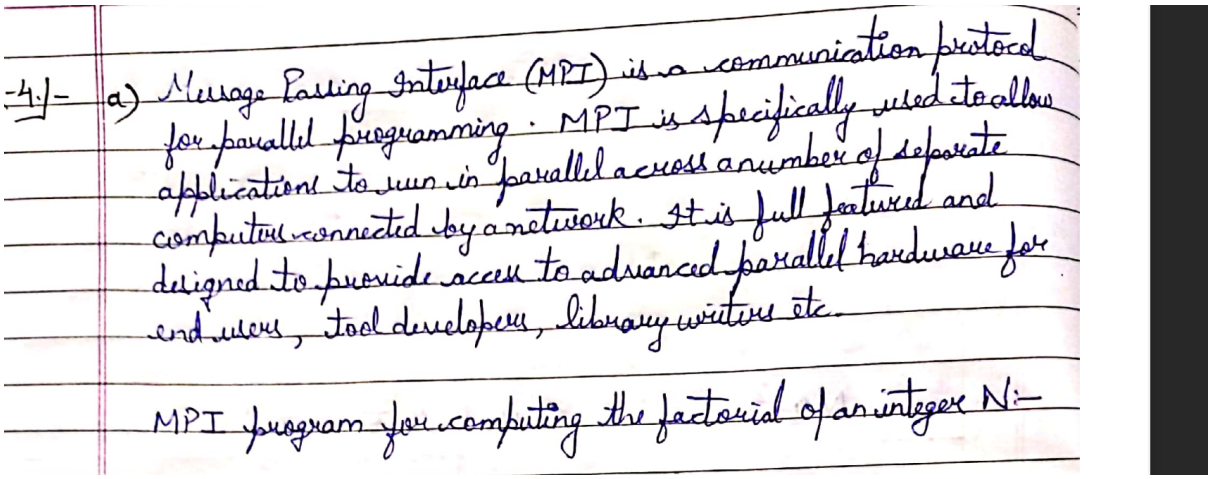
At each step n processes process log(n) lists in constant time O(1).
The parallel execution time is O(logn) and there are n processes. Total time complxity is θ(nlogn).
This complexity did not change from the sequential one but we have a achieved an algorithm that can run on parallel processors, meaning it will execute much faster at a larger scale.

Space complexity is O(logn).

- cpu vs gpu



- mpi program for factorial -
-



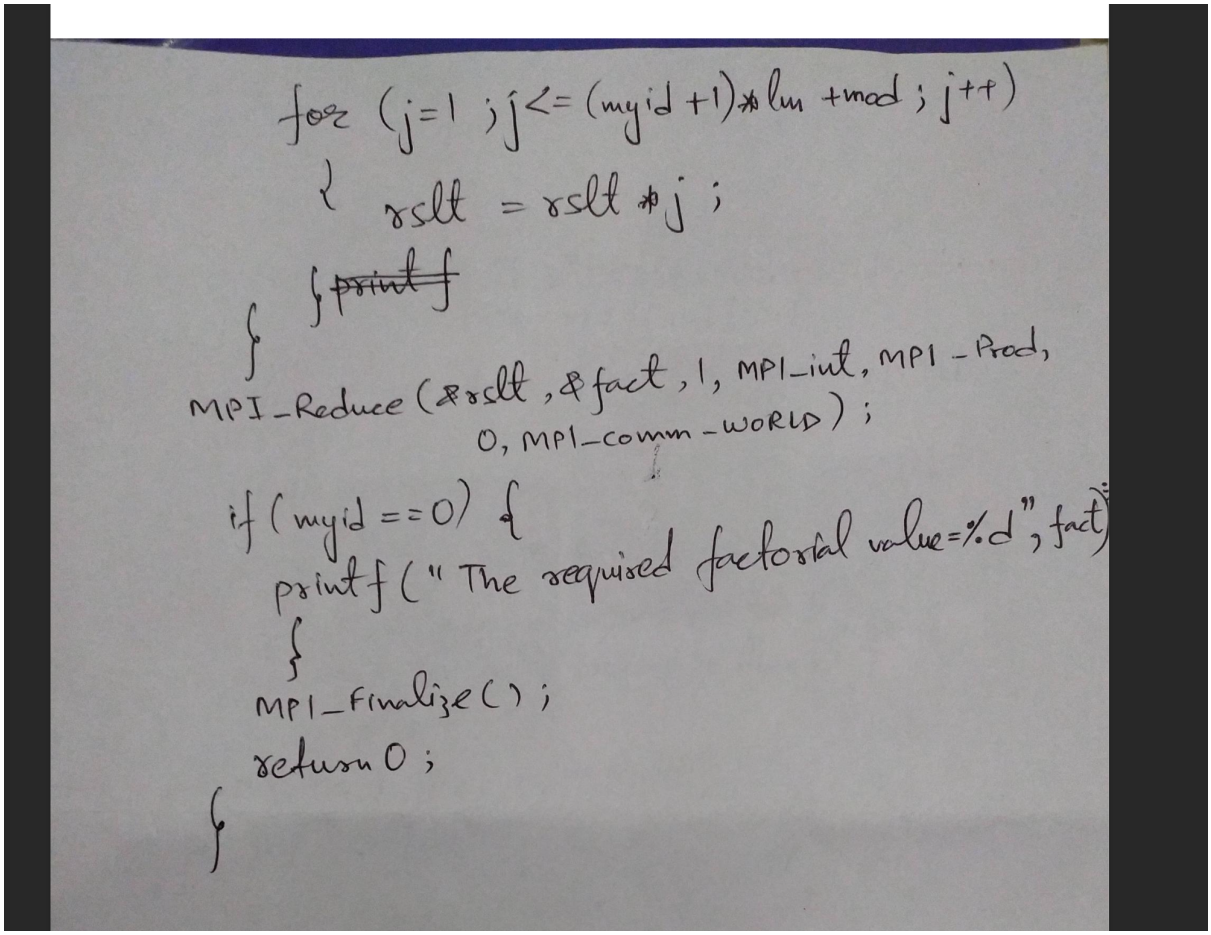
4.)- a) Message Passing Interface (MPI) is a communication protocol for parallel programming. MPI is specifically used to allow applications to run in parallel across a number of separate computers connected by a network. It is full featured and designed to provide access to advanced parallel hardware for end users, tool developers, library writers etc.

MPI program for computing the factorial of an integer N:-

Qno.4) Ans MPI program to find factorial of a given integer(n):

```
#include <stdio.h>
# include <mpi.h>
  int main (int argc , char **argv)
   {
     int  myid, numprocs, i, n, lm, j, mod;
     int  fact,rslt=1;
        MPI_Init (&argc, &argv);
        MPI_comm_size(MPI_comm_WORLD, &numprocs);
        MPI_comm_rank(MPI_comm_WORLD, &myid);
    int  unused attribute ((unused));
    if (myid ==0) {
          printf ("Enter the No.=> \n");
          unused = scanf ("%d", &n);
      }
  MPI_Bcast (&n, 1, MPI_INT, 0, MPI_comm-WORLD);
   lm = n/numprocs ;
```

```
   if (myid ==0) {
         printf ("Enter the No.=> \n");
         unused = scanf ("%d", &n);
     }
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_comm-WORLD);
 lm = n/numprocs ;
  mod = n % numprocs;
for (i=myid*lm+1 ; i< myid*lm+lm; i++)
 { rslt =rslt * i;
    if (mod != 0 && myid == numprocs-1)
    {
```



- mobile agent

Shubham Gupta
1713CS017

DATE: __/__/____
PAGE _____

Q-5] a) Mobile Agent

A mobile agent is a transportable program or object. In this paradigm, an agent is launched from an originating host. The agent then travels autonomously from host to host according to an itinerary that it carries. At each stop, the agent accesses the necessary resources or services and performs the necessary task to accomplish its mission.

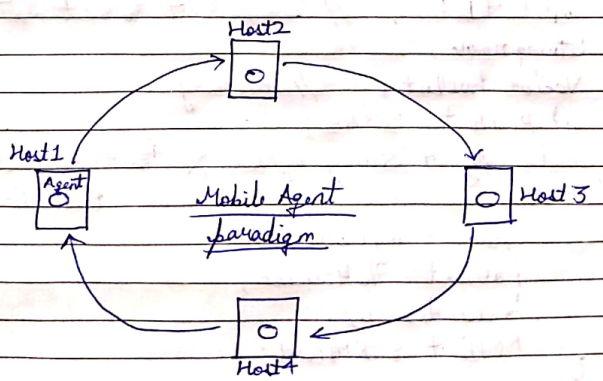
Agent carries following data items:

→ Identifying information: Information that allows the agent to be identified.

→ Itinerary: A list of addresses of the host that the agent is to visit.

→ Logic code to form its task.

Examples: Mobile Agent systems include Concordia system and Aglet system.

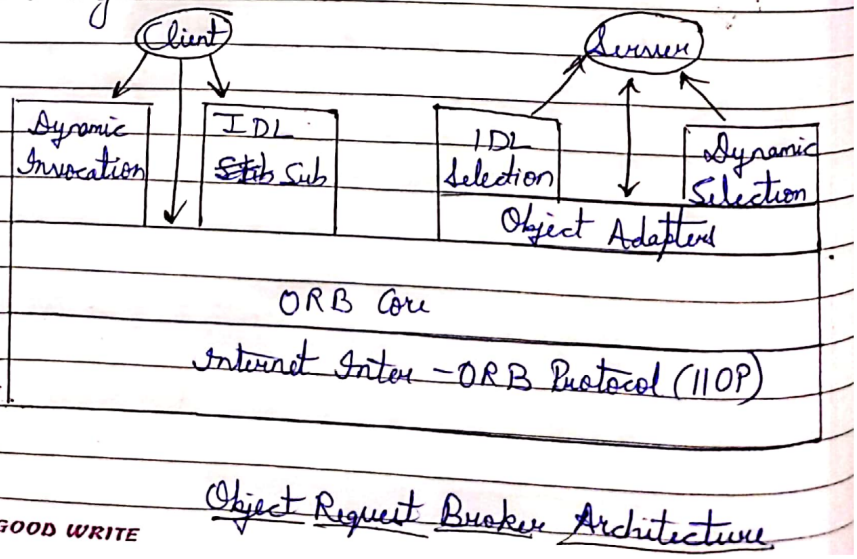


Program for Mobile Agent:-

- object request broker

b) Object Request Broker:

It is the programming that acts as a "broker" in between a client request for a service from a distributed object or component and the completion of that request. Having ORB support in a network means that a client program can request a service without having to understand where the server is in a distributed network or exactly what the interface to the server program looks like. Components can find out about each other and exchange interface info. as they are running.



GOOD WRITE

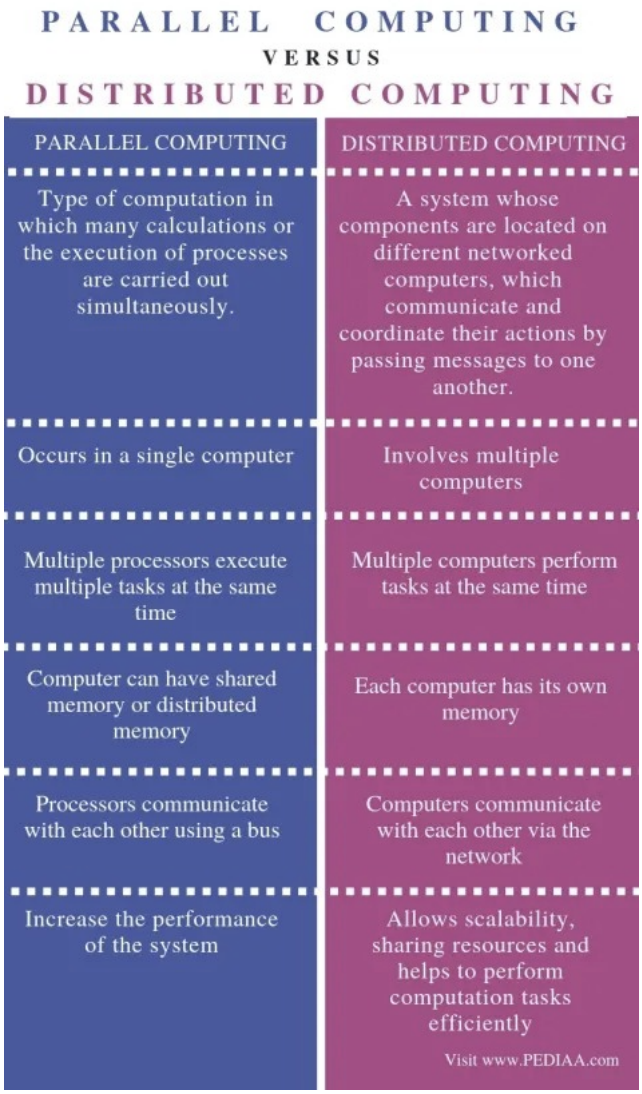
Object Request Broker Architecture

- matrix logical clock -

A matrix logical clock is a concept used in distributed systems to help keep track of the order in which events occur across multiple computers. In a distributed system, multiple computers may be working on different tasks and updating their own local data stores. In order to maintain consistency and prevent conflicts, it is important to have a way to determine the order in which events occurred.

A matrix logical clock uses a matrix to keep track of the logical time of events on each computer in the system. The matrix is updated whenever an event occurs on a particular computer, and the updated matrix is then sent to all other computers in the system. This allows each computer to maintain an up-to-date view of the logical time of events on all other computers in the system.

- distributed vs paralllel -


PARALLEL COMPUTING
VERSUS
DISTRIBUTED COMPUTING
PARALLEL COMPUTING
DISTRIBUTED COMPUTING
Type of computation in which many calculations or the execution of processes are carried out simultaneously.
A system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.
Occurs in a single computer
Involves multiple computers
Multiple processors execute multiple tasks at the same time
Multiple computers perform tasks at the same time
Computer can have shared memory or distributed memory
Each computer has its own memory
Processors communicate with each other using a bus
Computers communicate with each other via the network
Increase the performance of the system
Allows scalability, sharing resources and helps to perform computation tasks efficiently
Visit www.PEDIAA.com

MPI-OpenMP Matrix Multiplication

```
MPI_Scatter(A, blksz*N, … );          // Scatter input matrix A
    MPI_Bcast(B, N*N, … );            // Broadcast  input matrix B
    for(i = 0 ; i < blksz; i++) {
        #pragma omp parallel for private (sum,k)
        for (j = 0 ; j < N ; j++) {
            sum = 0;
            for (k = 0 ; k < N ; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    MPI_Gather(C, blksz*N, … );
```

Parallelize i loop into partitions among processes on computers with MPI

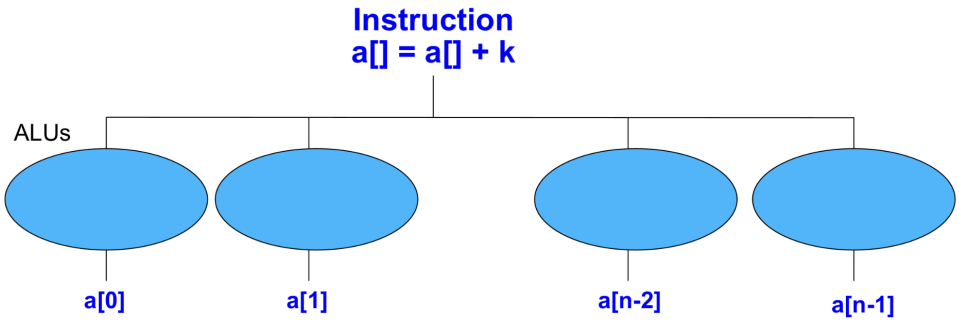
Parallelize j loop on each computer into partitioned using OpenMP

Simply add this one statement to MPI code for matrix multiplication

14

SIMD (Single Instruction Multiple Data) model

Also know as data parallel computation.
One instruction specifies the operation:



Very efficient of this is what you want to do. One program.
Can design computers to operate this way.

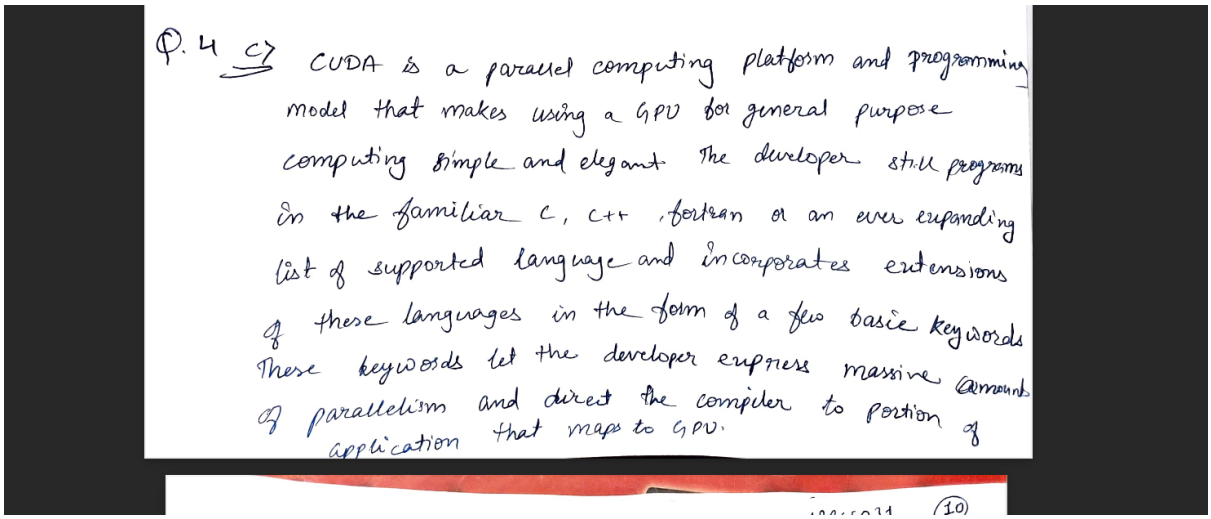
3

Summary - 1

- We have looked at a wide range of paradigms for distributed applications.
- The paradigms presented were:
  - Message passing
  - Client-server
  - Message system: Point-to-point; Publish/Subscribe
  - Distributed objects:
    - Remote method invocation
    - Object request broker
    - Object space
  - Mobile agents
  - Network services
  - Collaborative applications

- distributed paradigms
- cuda program matrix
- pram program for maximum of all numbers
- openmp program for pi

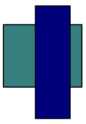
cuda for matrix multiplication -


Q. 4 c) CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant. The developer still programs in the familiar C, C++, fortran or an ever expanding list of supported language and incorporates extensions of these languages in the form of a few basic keywords. These keywords let the developer express massive amounts of parallelism and direct the compiler to portion of application that maps to GPU.

CUDA Program using C for matrix multiplication

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define BLOCK_SIZE 16

__global__ void mm_kernel(float* A, float *B, float* C,
                          int n)
{
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  if (row < n && col < n) {
     for(int i = 0; i<n ; ++i) {
        C[row*n + col] += A[row*n+i] *
                          B[i*n + col];
     }
  }
}
int main() {
     int nx = 1000, ny = 1000;
     dim3 blockDim(16,16);
     int gx = nx%blockDim.x == 0 ? nx/blockDim.x :
                                   nx/blockDim.x+1;
     int gy = ny%blockDim.y == 0 ? ny/blockDim.y :
                                   ny/blockDim.y+1;
     dim3 gridDim(gx,gy);
     mm_kernel <<< gridDim, blockDim >>> (d_a, d_b, d_c, n);
     __shared__ float C[size];
     __shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];
     Msub[threadIdx.y][threadIdx.x] = M[Tidy*width+TidX];
     __syncthreads();
}
```

Basic CUDA program structure



int main (int argc, char **argv) {

1. Allocate memory space in device (GPU) for data
2. Allocate memory space in host (CPU) for data

3. Copy data to GPU

4. Call “kernel” routine to execute on GPU
(with CUDA syntax that defines no of threads and their physical structure)

5. Transfer results from GPU to CPU

6. Free memory space in device (GPU)
7. Free memory space in host (CPU)

return;

}

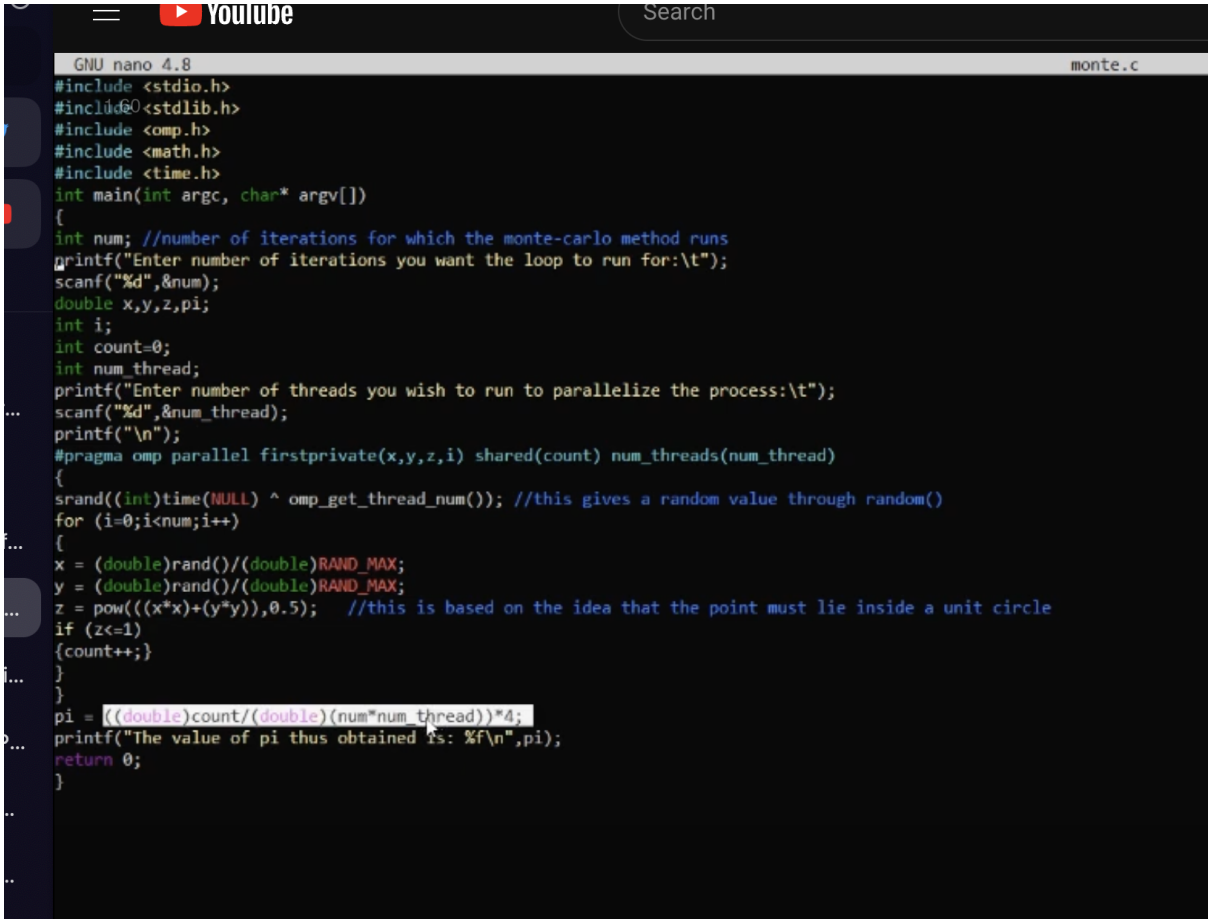
8

```
// This program computes a simple version of matrix multiplication
// By: Nick from CoffeeBeforeArch

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::generate;
using std::vector;

__global__ void matrixMul(const int *a, const int *b, int *c, int N) {
  // Compute each thread's global row and column index
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  // Iterate over row, and down column
  c[row * N + col] = 0;
  for (int k = 0; k < N; k++) {
    // Accumulate results for a single element
    c[row * N + col] += a[row * N + k] * b[k * N + col];
  }
}

int main() {
  // Matrix size of 1024 x 1024;
  int N = 1 << 10;

  // Size (in bytes) of matrix
  size_t bytes = N * N * sizeof(int);

  // Host vectors
  vector<int> h_a(N * N);
  vector<int> h_b(N * N);
  vector<int> h_c(N * N);

  // Initialize matrices
  generate(h_a.begin(), h_a.end(), []() { return rand() % 100; });
  generate(h_b.begin(), h_b.end(), []() { return rand() % 100; });

  // Allocate device memory
  int *d_a, *d_b, *d_c;
```

```
  cudaMalloc(&d_a, bytes);
  cudaMalloc(&d_b, bytes);
  cudaMalloc(&d_c, bytes);

  // Copy data to the device
  cudaMemcpy(d_a, h_a.data(), bytes, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b.data(), bytes, cudaMemcpyHostToDevice);

  // Threads per CTA dimension
  int THREADS = 32;

  // Blocks per grid dimension (assumes THREADS divides N evenly)
  int BLOCKS = N / THREADS;

  // Use dim3 structs for block  and grid dimensions
  dim3 threads(THREADS, THREADS);
  dim3 blocks(BLOCKS, BLOCKS);

  // Launch kernel
  matrixMul<<<blocks, threads>>>(d_a, d_b, d_c, N);

  // Copy back to the host
  cudaMemcpy(h_c.data(), d_c, bytes, cudaMemcpyDeviceToHost);

  // Check result
  verify_result(h_a, h_b, h_c, N);

  cout << "COMPLETED SUCCESSFULLY\n";

  // Free memory on device
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

openmp definition -

OpenMP is a programming interface for writing parallel applications that can run on shared memory systems. It allows a programmer to write code that can be executed by multiple threads or processors simultaneously, which can improve the performance of certain types of applications. OpenMP is based on the C and C++ programming languages, but it can also be used with other languages that are compatible with C and C++. OpenMP is an open standard, which means that it is freely available for anyone to use.


YouTube
Search
GNU nano 4.8
monte.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <time.h>
int main(int argc, char* argv[])
{
int num; //number of iterations for which the monte-carlo method runs
printf("Enter number of iterations you want the loop to run for:\t");
scanf("%d",&num);
double x,y,z,pi;
int i;
int count=0;
int num_thread;
printf("Enter number of threads you wish to run to parallelize the process:\t");
scanf("%d",&num_thread);
printf("\n");
#pragma omp parallel firstprivate(x,y,z,i) shared(count) num_threads(num_thread)
{
srand((int)time(NULL) ^ omp_get_thread_num()); //this gives a random value through random()
for (i=0;i<num;i++)
{
x = (double)rand()/(double)RAND_MAX;
y = (double)rand()/(double)RAND_MAX;
z = pow(((x*x)+(y*y)),0.5); //this is based on the idea that the point must lie inside a unit circle
if (z<=1)
{count++;}
}
}
pi = ((double)count/(double)(num*num_thread))*4;
printf("The value of pi thus obtained is: %f\n",pi);
return 0;
}