Name: Shahrukh Khan

B. Tech. Computer Engineering

VIIth Semester

R. No. 18BCS031

Paper code: CEN-704

Parallel and Distributed Computing

Date: 30-12-2021

Time: 10.00 A.M.

Q. 1'

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

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 $= 16789504 / 520384$

$= 32.26$ An

iii) The parallel program is

```
PAR for ( L=1 ,  L<=64 , L++) {
    for (I = (L-1)*64 + 1 ; I <= L+64 ; I++) {
        sum [I] = 0
```

```
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 $= 16789504 / 262336$

$= 64$

Ans

Q. 2b)

bernstein conditions are

$W(S1) \cap R(S2) = \{\}$

$W(S2) \cap R(S1) = \{\}$

$W(S1) \cap W(S2) = \{\}$

$W(S1) = \{A\}$

$R(S1) = \{B, C\}$

$W(S2) = \{C\}$

$R(S2) = \{B, D\}$

$W(S3) = \{S\}$

$R(S3) = \{\}$

$W(S4) = \{I, S\}$

$W(S4) = \{S, X, I\}$

$W(S5) = \{E\}$

$R(S5) = \{S, C\}$

The bernstein conditions are satisfying the following

→ S1, S3

→ S2, S3

→ S2, S4

→ S1, S4

Unsatisfied conditions are

→ S1, S2

$R(S1) \cap W(S2) = \{C\}$

→ S2, S5

$W(S2) \cap R(S5) = \{C\}$

→ S3, S4

$W(S3) \cap W(S4) = \{S\}$

→ S3, S5

$W(S3) \cap R(S5) = \{S\}$

→ S4, S5

$W(S4) \cap R(S5) = \{S\}$

→ S1 , S5

$R(S1) \cap W(S5) = \{C\}$

$(S1 \parallel S3)$, $(S2 \parallel S3)$, $(S2 \parallel S4)$,

$(S1 \parallel S4)$

P1 = A = B+C

S = 0

P2 = C = B * D

for (I = 0 ; I <= 100 ; I++)

S := ~~S+X[I]~~ S + X[I]

P3 = if (S > 1000) C = C * 2

Q. 3 a) SIMD algorithm for NxN 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
```

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

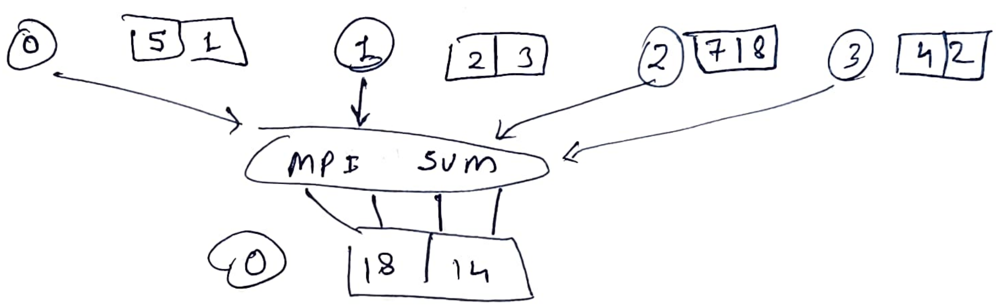
```
#include <stdio.h>
#include < stdlib.h >
#include < omp.h >
int main (void ) {
  int n = 10 ;
  int factorial [n];
  factorial [1] = 1;
  int * Proda ;
  #Pragma omp parallel
  {
     int ithread = omp_get_thread_num();
     int nthreads = omp_get_num_threads();
     #pragma omp single
      {
         Proda = malloc (nthreads * size of *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< (ithreads + 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]);
}
```

Q. 4 a)

i) MPI reduce:-

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

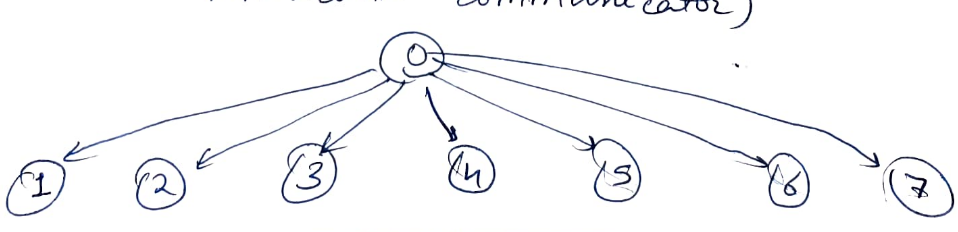
e.g



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

ii) MPI_Broadcast

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

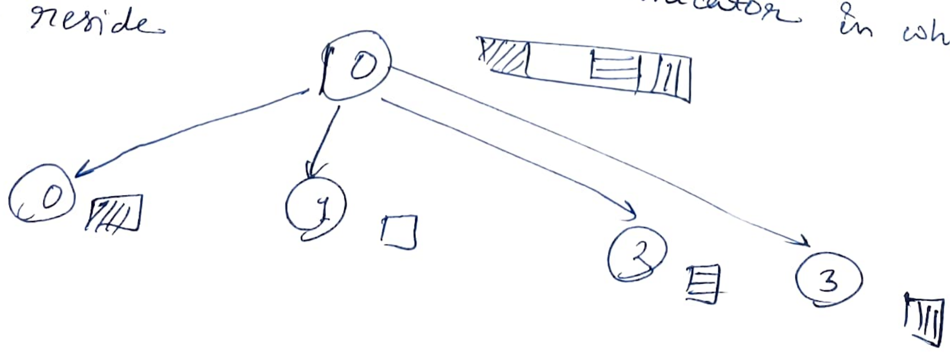


In this process, process 0 is the root process and it has initial copy of data. All other process receive the copy of data.

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 )
```

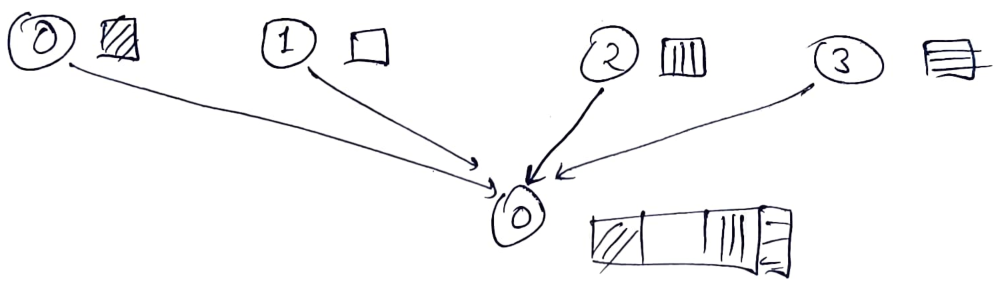
The first parameter send-data is array of data that resides on root process. The second and third parameter send_count and send_datatype dictate how many elements of a specific datatype will go to each process. If send_count is 1 and send_data is MPI_INT, then process zero gets first integer of array, 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)
```



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 parallelisms and direct the compiler to portion of application that maps to GPU.

CUDA Program using C for matrix multiplication

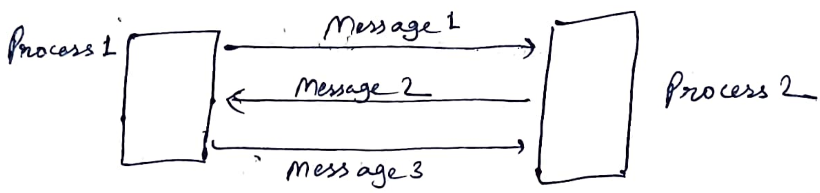
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define BLOCK_SIZE 16

__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 [size];
    __shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];
    Msub[threadIdx.y][threadIdx.x] = M[TidY * width + TidX];
    __syncthreads();
}
```

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

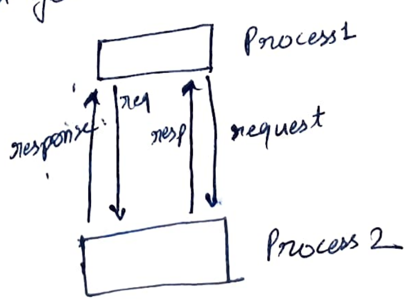


⇒ Client Server Paradigm :- The server acts as a service provider, the client issues the request and wait for response from the server. Here server is dump machine. Until client make a call, server doesn't communicate. Many internet services are client-server applications.

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 act 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

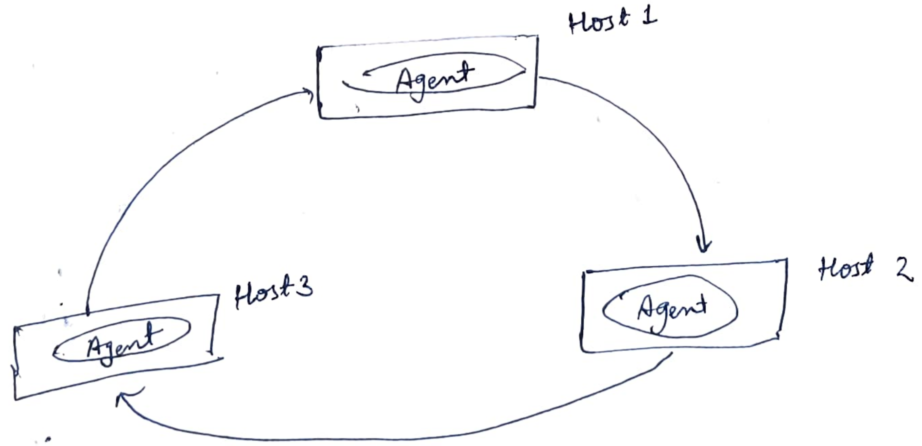
⇒ Distributed Object Paradigm:-

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 :-

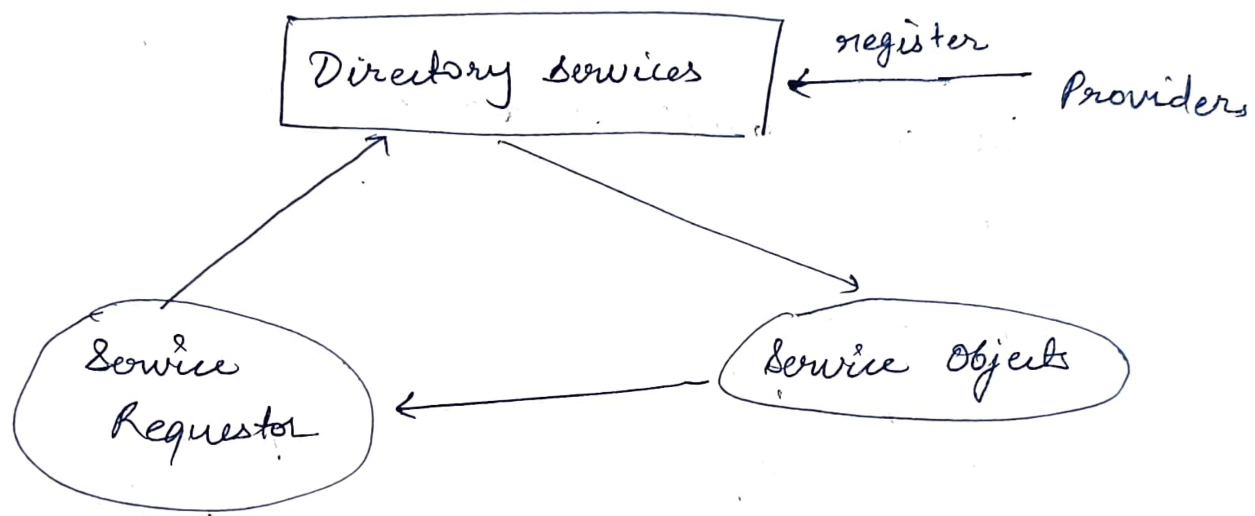
Mobile Agent starts from originated host and transports over host to host. At each host, the agent can access the services or resources to complete the mission



⇒ Network service Paradigm :-

All the service objects are register with global directory service. If process wants, a service can contact directory service at runtime. Requestor is provided a reference, using which process interact with service. Services are identified by the global unique identifier

example. Java Jini



⇒ Collaborative Application Paradigm :

Processes participate in a collaborative session as a group. Each participating process may contribute input to part or all of the group.

Types :- → Message based groupware paradigm

→ Whiteboard based groupware paradigm