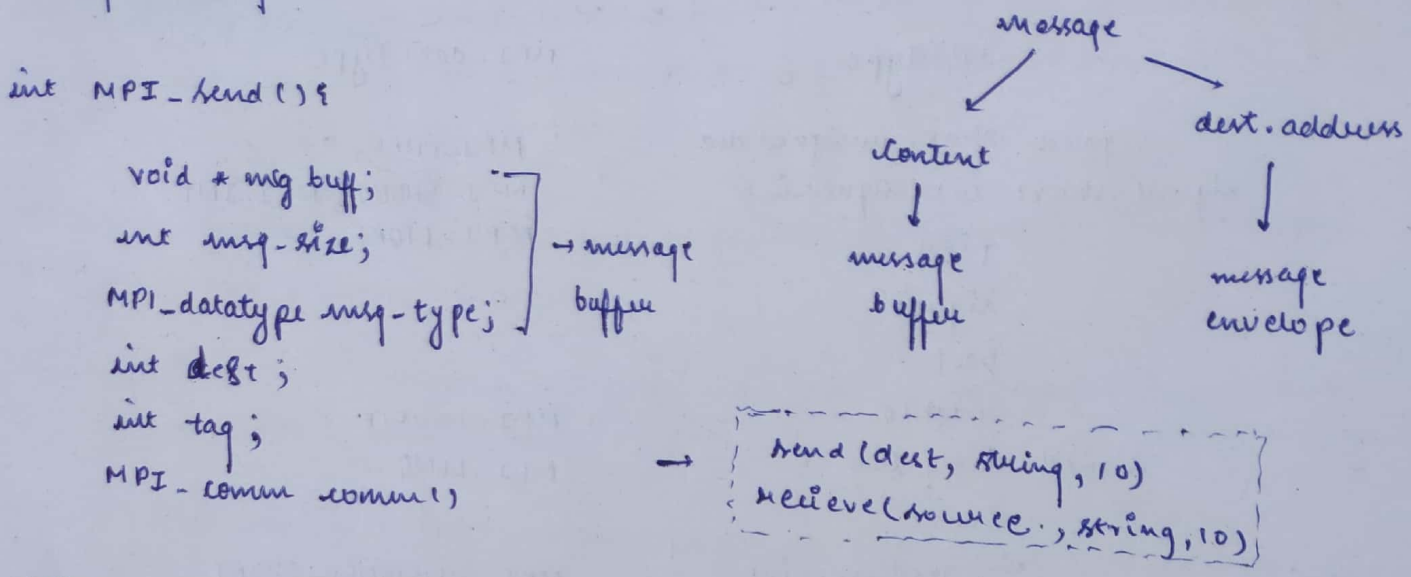
Message Passing →



⇒ MPI_Send has 6 parameters, The first three parameters constitutes of message buffer and signifies the content of the message.

(i) The first one is msg-buff-p,
the pointer to the block of memory containg the content of the message. the msg address could be the any address in the ~~mess~~ sender's address and refers to the starting of the data.

(ii) The second argument is msg-size, it is also called the message count and specific the no. of occurrences of data item of the message data type starting at the msg-add, count argument can be used to group continuous array element into the group.

(iii) The MPI datatype parameter is used to support heterogeneous computing, and allow messages from non-contiguous memory location.

MPI_Send (dest, ~~string~~, 10, MPI_CHAR, -----)
MPI_recieve(B, 10, MPI_char, ------)

Here, 10 no longer signifies the mesage length in bytes but it specifies the occurences of data items of the message type MPI_CHAR. Thus data type MPI_char is implemented as 1 byte in sending machine but 2 bytes in recieving machine, The implementation is responsible for the conversion b/w 8 bits and 16-bits format.

Basic MPI-data types →

C-data type	MPI-data type
signed char, unsigned cha	MPI_CHAR,
signed short int, signed int	MPI-SHORT, MPI_INT
float	MPI-FIOAT
signed	
bool	
double	MPI-DOUBLE
signed long int	MPI-LONG
short	
unsigned short int	MPI_UNSIGNED-SHORT
unsigned long int	MPI -UNSIGNED-LONG
long double	MPI-LONG-DOUBLE
	MPI-BYTE → when messages is sent in bit
	MPI-PACKED
	↳ non-continuous data to sent

void.

int MPI-Pack(

void * in_buff,
int in_buf_count,
MPI_Datatype datatype,
void * pack_buf;
int pack_buf_size,
int * position_p;
MPI_comm comm);

↳ to pack the data into the contiguous memory spaces

int MPI_Unpack(

void * pack_buf,
int pack_buf_size,
int * position_p,
void * out_buf,
int out_buf_count,
MPI_Datatype datatype,
MPI_comm comm);

↳

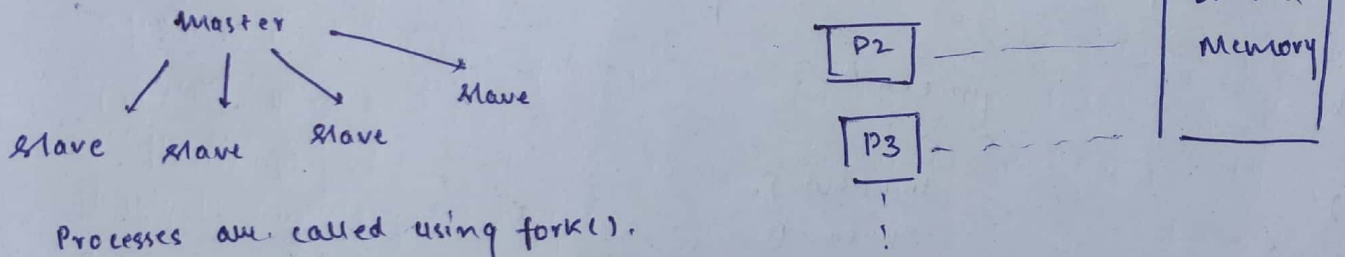
MPI pack takes the data to be packed, and packs it into continuous buffer, the position_p element keeps track in where we are in our continuous buffer. When the function is called it refers to the first available location in the buffer before data to packed is added. to

When function returns, it refers to first available location is the packed buffer after data to be packed is added.

MPI unpack reverses the process, it takes the data in contiguous buffer and unpacks it into unpacked data.

When the function is called position_p refers to first location in continous buffer that hasn't been ~~up~~ unpack and it returns the *position_p next location in continuous buffer after the data that was just unpacked.

Shared memory Program →



Processes are called using fork().
and are terminated using join().

- In shared memory programming, programmers view their program as a collection cooperating processes, excessing a common pool of shared variables.
- Each processor may not have private data memory, A common program and data are stored in the main memory and are shared by all the processors.
- Each processor can be assigned different part of the program stored in memory to execute with data, also stored in the specified locations.

Process creation →

- The main program creates sperate processes for each processor and allocates them along with the information on the location, where data are stored for each process.
- Each processor computes independently.
- when all processors completes their assign tasks they have to rejoin with the main program.

- The main program will execute; after all processes created by it finished.

Statements for parallel execution are added in the programming lang to enable creation of processes and for waiting for them to complete.

Two statements that are used for this purpose →

(1) fork() → which is used to create a process.

(2) Join() → which is used when the invoking process needs the result of the invocked processe to continue.

```
Process X
{
   ---
   ---
   ---
   fork (Y)
   ---
   join (Y)
   end (X)
}
```

```
Process Y
{
   ---
   ---
   ---
   ---
   end (Y);
}
```

The one of the main problem is inconsistency which can be resolved using semaphor -es.

```
Process A
{
   fork (B)
   lock (sum)
   sum += A
   unlock (sum)
   join (B)
   end (A)
}
```

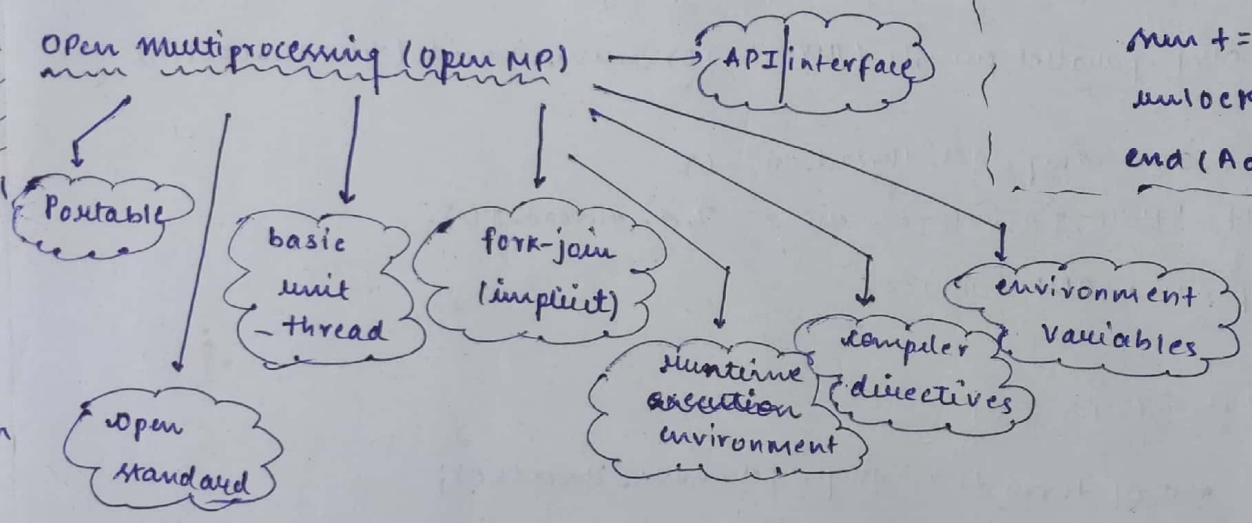
```
procen B
{
   lock (sum);
   sum += B
   unlock (sum);
   end (B);
}
```

```
//main program  P0 - master, P1 → slave
begin (main)
   global sum, Array[1:N]
   for i = 1 to N
      read A[i]
   end (for)
   sum = 0;
```

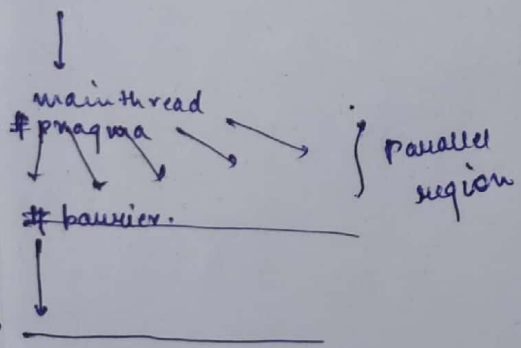
```
//Processor 1 executes
fork(1) Add_array (input A[N/2+1 ; N], output sum);
.
//processor 0 executes
Add_array (input A[1: N/2], output.sum)
join(1)
write( sum );
end (main);
```

```
Add_array [input: A[p:M],
           output ; sum)'
begin Add_array)'
sum_local = 0;
for i = p to m :
    sum_local += A[i];
endfor
lock(sum)
sum += sum_local ;
unlock(sum);
end (Add_array);
```

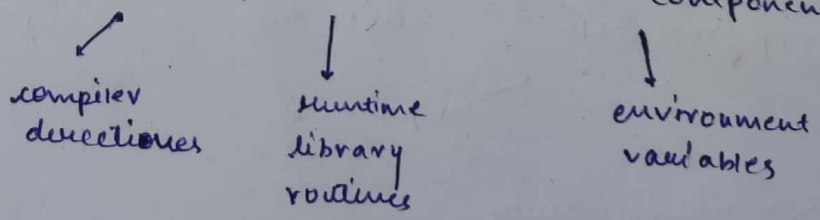
15/02/21



- It is a portable standard for programming shared memory parallel computers.
- Like MPI, open MP is explicit programming model.
- Open MP allows the programmer the facility of incremental parallelization



- Open MP comprises of three distinct components



- The central entity in an open MP program is a thread not a process
- A open MP program creates multiple coperating threads, which runs on multiple processor.
- The execution begins with the single thread called master thread, which executes the program sequentially until it encounters the first parallel construct.

- At the parallel construct, the master thread, creates a team of threads consisting of number of new threads called slaves.
- The fork and join operations are implicit
- At the end of the parallel region, there is a implicit barrier synchronization and only the master thread continues to further execution.

```
#include <stdio.h>
#include <omp.h>
int main(){
    int no.of threads, thread_ID, THREAD_COUNT=4;
    #pragma omp parallel private (THread_ID) num_threads (THREAD_COUNT)
    {
        thread_ID = omp_get_thread_num()
        Printf("Hello, I am thread number %d", thread_ID);
    }  #pragma OMP barrier;
    #pragma
        if (thread_ID==0)
        {
            no. of threads = omp_get_num_threads();
            printf("Master - My team, has %d threads", no_of_threads);
        }
    }
    return 0;
}
```

⇒ Scope of variable → shared
→ private

Loop-carried dependency

X+1 = Result is dependent on the result of previous state

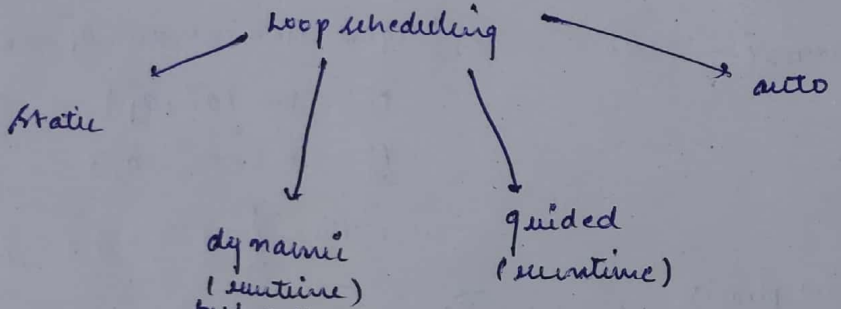
```
for(i= 1 to 10){
    a[i] = a[i]+3
    b[i] = b[i]+a[i-1]
}
```

→ these types of loops cannot be parallelized by open MP.

#pragma omp parallel for.

Loop scheduling in OpenMP

- It is possible to incrementally parallelize a sequential program, that has many for loops by successfully using OpenMP parallel for directive for each loop.
- However, this loop has 2 restrictions.
  - The total no. of iterations must be known in advanced
  - The iterations must be independent of each other.



Syntax → schedule (type , chunk_size);

directives →

- barrier
- critical → multiple statements to restrict
- automatic → single statements to lock (restrict).
- nowait → to avoid synchronization delay at time of joining.
- reduction → (operator, variable).

⇒ Environment variables → or manipulate

Variables that are used to at the time of runtime.

Omp_dynamic, Omp_schedule

Omp_stacksize, Omp_no_of threads.

⇒ P_threads =

Odd even Transposition sort :-> compare exchange (P_i^o, P_j^e)

carried out in 2 phases
odd phases
and even phase

compare exchange operation

(1) Let say there are 2 processors P_i and P_j having 2 elements a_i and a_j. so in first steps both processors exchange their values such that both of them contains both the elements.

(2) Now, left processor contains the smallest element and right processor contains the largest element out of them.

$P_i^o = \min(a_i^o, a_j^o)$

$P_j^e = \max(a_i^o, a_j^o)$

(3) odd phase → (first phase)

→ $\langle a_1, a_2 \rangle$ in odd phase elements of odd number/position is compared to its right element and compare exchange operation is performed
$\langle a_3, a_4 \rangle$
!

(4) even phase → (second phase)

$\langle a_2, a_3 \rangle$ same with even position elements
$\langle a_4, a_5 \rangle$
!

for n number of times

n/2 odd phase

n/2 even phase

⇒ PRAM operations (sorting).

sort by enumeration

↳ each element compares itself with other elements.

$S = \langle S_1, S_2, \ldots S_n \rangle$

$R = \langle r_1, r_2, \ldots r_n \rangle$

↳ it is initially initialized to zero

Rank r_i of an element s_i is the number of elements smaller than s_i in the sorted sequence.

↳ The final position of the element s_i in the sorted sequence is given by placing the element s_i in position $r_i + 1$.

CRCW sorting Implementation

```
for i = 1 to n do in parallel
    for j = 1 to n do in parallel
        if si > sj or (si == sj and i > j) then
            Pi,j writes 1 to ri
        end if
    end for
end for

for i = 1 to n do in parallel
    Pi,1 puts si in (rj + 1) position of s
end for
```

P_1 (1,1)	P_2 (1,2)	P_3 (1,3)	P_4 (1,4)
P_5 (2,1)	P_6 (2,2)	P_7 (2,3)	P_8 (2,4)
P_9 (3,1)	P_{10} (3,2)	P_{11} (3,3)	P_{12} (3,4)
P_{13} (4,1)	P_{14} (4,2)	P_{15} (4,3)	P_{16} (4,4)

$s = \langle 12, 5, 7, 1 \rangle$

Here concurrent read and write operations are performed.

(12,1) (12,2) (12,5) (12,7)

(12,1) ↩ no operation

(12,2) ↩ no operation

(12,5) ↩ P_2 changes to 1 in r_1

(12,7) ↩ "

(12,1) ↩ "

$r_1 = \boxed{3 \mid 1 \mid 2 \mid 0}$ ←

so this rank matrix can be used to sort the array.

after all the operations on r_i we add them all

CREW sorting

```
for i = 1 to n do in parallel        ← concurrent read
    for j = 1 to n                   ← exclusive write
        if (sj > sj) or (si == sj and i > j) then
            Pi,j adds 1 to ri
        end if
    end for
endpar
Pi+1 puts si in (ri+1) positions of s
end for
```

Here we only use n processors.

parallelly → s_1	s_2	s_3	s_4
s_1	s_1	s_1	s_1
s_2	s_2	s_2	s_2
s_3	s_3	s_3	s_3
s_4	s_4	s_4	s_4

This occurs serially

EREW sorting

```
for i = 1 to n do in parallel
    for j = 0 to n-1
        K = (i + j) mod n
        if (Si > SK) or (Si == SK and i > K) then
            Pi adds 1 to ri
        end if
    end for
    Pi put Si in (ri + 1) position of
end for
```

Matrix multiplication (PRAM)

```
for i = 1 to m do
    for j = 1 to k do
        Cij = 0
        for s = 1 to n do
            Cij = Cij + ais x bsj
        end for
    end for
end for
```

← Simple matrix multiplication

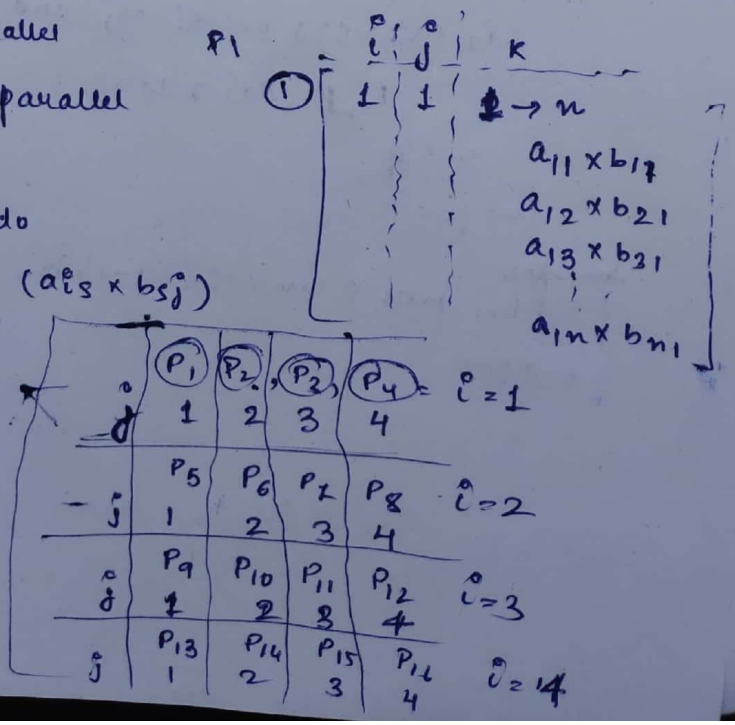
$[m \times k] \times [k \times n]$

CREW → n^2
EREW → n^2
CRCW → n^3
ERCW → ✗

$(n \times n)$

for CREW →

```
for i = 1 to n do in parallel
    for j = 1 to n do in parallel
        Cij = 0
        for k = 1 to n do
            Cij = Cij + (ais x bsj)
        end for
    end for
end for
```



EREW

```
for i=1 to n do in parallel
    for j=1 to n do in parallel
        cij = 0
        for K=1 to n do
            lK = ((i+j+K) mod n) + 1
            cij = cij + ai,lK * blK,j
        end for
    end for
end for
```

$c_{11} += (a_{14} * b_{41})$ (11)

	i	j	K	lK	
P1	1	1	1	4	$a_{14} * b_{41}$
			2	1	$a_{11} * b_{11}$
			3	2	$a_{12} * b_{21}$
			4	3	$a_{13} * b_{31}$
P2	1	2	1	1	$a_{11} * b_{12}$
			2	2	$a_{12} * b_{22}$
			3	3	$a_{13} * b_{32}$
			4	4	$a_{14} * b_{42}$
P3	1	3	1	2	$a_{12} * b_{23}$
			2	3	$a_{13} * b_{33}$
			3	4	$a_{14} * b_{43}$
			4	1	$a_{11} * b_{13}$

CRCW →

```
for i = 1 to n do in parallel
    for j=1 to n do in parallel
        for K = 1 to n do in parallel
            cij = 0
            cij = aik * bkj
        end for
    end for
end for
```

Unit-3

Inst. Level Parallelism

Ideal conditions

(1) All instructions breakable into equal sized instruction : → integer additions
→ floating point number of add.

(2) Sequential program (locality of references)
↳ i.e. no branches.

(3) Switching time

(4) Independent of each ~~resources~~ others.

(5) Sufficient resources.

RISC →

Caches

I-C → memory → IR

D-C → Data → MDR

32 GPRS

ADD R1 R2 R1

Types of instructions →
- R - opcode field, three operands
- M type - opcode field, 2 operands, memory address
- J type - opcode, memory address
  ↳ JMP 201

(2) program counter

(3) only 2 operations can access your memory.

(4) ALU at any single time →
- arithmatic
- logical

} clock cycle.

Instruction execution cycle

(1) Fetch instruction

(2) Decode and fetch register

(3) Execute and calculate ~~execution~~ effective address.

(4) memory Access

(5) Storage in the memory.