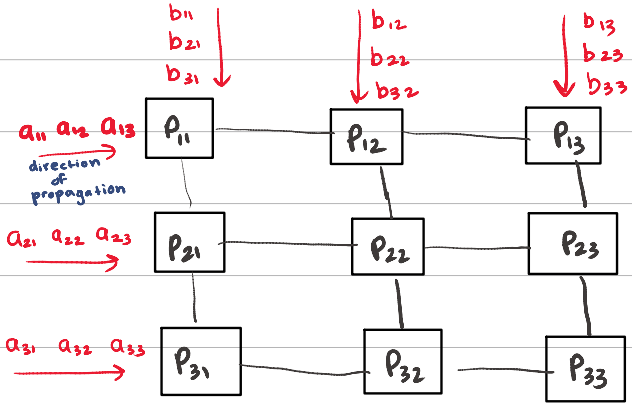
Date: 28/8/23

* Matrix Multiplication

PRAM	CRCW	$O(n^2)$ processors	$O(n)$ time
	EREW	$O(n^2)$	$O(n)$
	CRCW	$O(n^3)$	$O(1)$



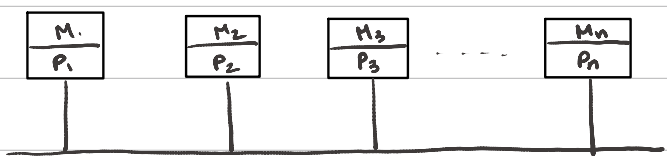
Propagation of values

$A^{3\times3} \times B^{3\times3}$

$O(n)$ time taken

* Bulk Synchronous Parallel Model (BSP)

it is an abstract parallel machine



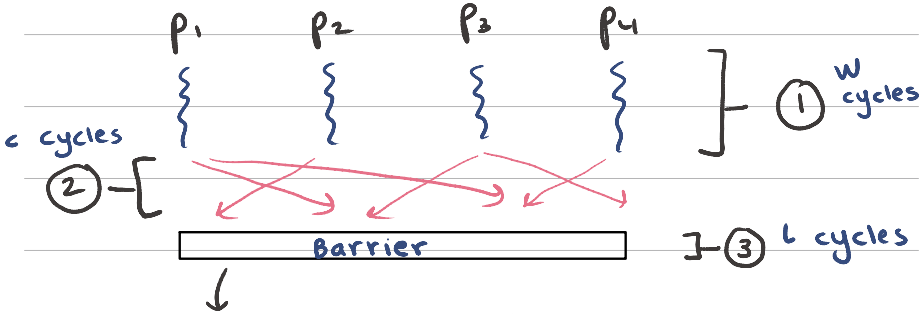
* Each processor has its own memory (Unlike PRAM → shared memory)

Date: /

Application

* there are n processes ; 1 per node
* each process executes one superstep sequentially

Superstep
1. Computation — W
2. Communication — C
3. Synchronization — L



- for synchronizing >2 processes
- check whether it is complete, etc

- time taken by P_i - $w_i + c_i + L_i$

w ≈ number of operations

L ≈ depends on hardware/algo

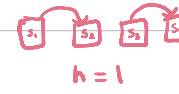
$c = gh$

- g: time/word (hardware dependant)
- h: max no. of words that can be sent/received by a processor in a superstep

Date: /

Example: $p = 4$, Adding n numbers $(n >> p)$

- As per model, # processes = 4 $(= p)$

(# Steps) →	Comp.	Comm.	Sync.
Superstep 1:	$\frac{N}{4}$ (Adding the $\frac{N}{4}$ nums)	1 (g cycles)  $h = 1$	1 (L cycles)
Superstep 2:	1 $(S_1 + S_2)$ and $(S_3 + S_4)$	1 (g cycles) 	1 (L cycles)
Superstep 3:	1 $S_{12} + S_{34}$	—	—

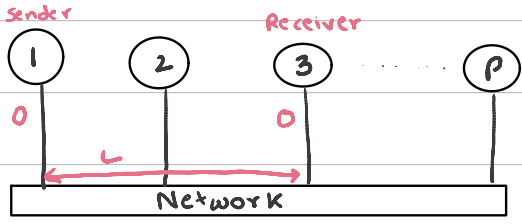
Total steps : $\left(\frac{N}{4} + 2\right) + 2g + 2L$

* Same problem on PRAM

- CRCW: $\left(\frac{N}{4} + 2\right)$
  - no comm.
  - can use sum CW
- EREW

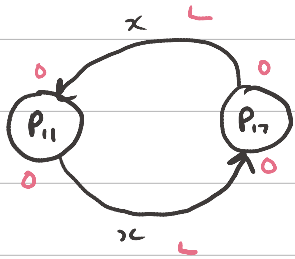
Date: 29/8/23

* Log P Model
  - P → # processors
  - gap
  - overhead (preparing the message - for communication)
  - latency



overhead: length of time that a processor is involved in the transmission/reception of msg.

Eg:



Total time

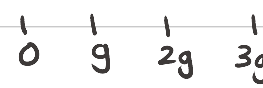
= 40 + 2L

latency: length of time taken to send a message from one processor to another.

Date: /

gap: min time interval between consecutive message termination/reception.

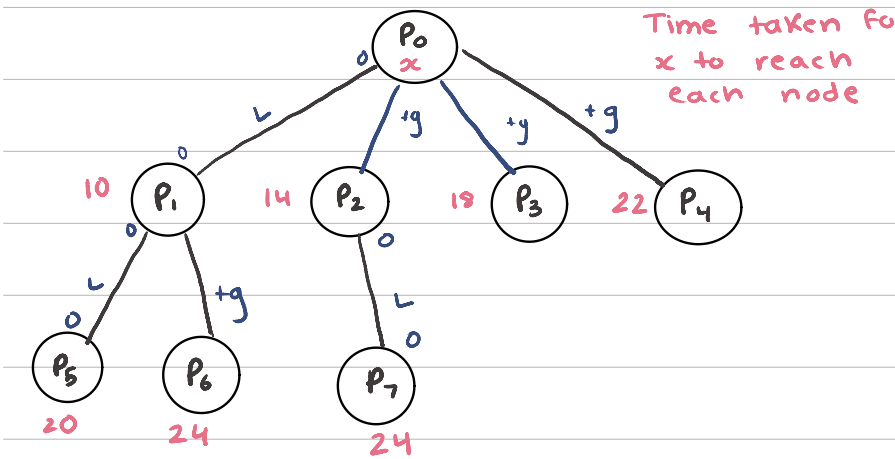
Eg: P_i pumps out messages at intervals



$$\frac{L}{g} = \#\text{ messages in the network}$$

* synchronization happens within sender-receiver pairs (not b/w ALL processors)

Example: $p = 8$, $L = 6$, $g = 4$, $o = 2$

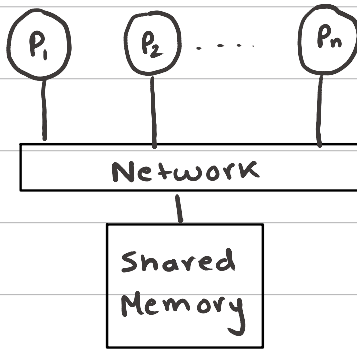


Date: /

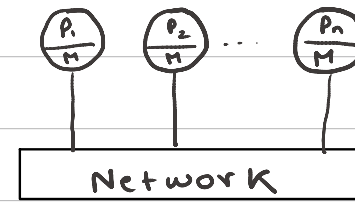
FOR EXAM

Chapters Completed - 1, 2, 7, 8, 11

* Parallel Programming



(Actual pgm. implementation)



* Focus for now

Example pseudo code:

Processor 0

Array A[1:N]

BEGIN

For i = 1 to N do

Read A[i]

End for

send (1, A[N/2 + 1], N/2)

(1 → processor; A[N/2 + 1] → variable; N/2 → length)

s

Processor 1

Array B[1 : N/2]

BEGIN

Receive (0, B[1], N/2)

set sum1 = 0

for i = 1 to N/2 do

sum1 = sum1 + B[i]

endfor

Date: /

```
set sum := 0
for i = 1 to N/2 do
    sum = sum + A[i]
end for
Receive (1, sum1, 1)
sum = sum + sum1
print (sum)
END
```

```
send (0, sum1, 1)
END
```

Note: * for every send, there MUST be a receive (and vice-versa)

* only one program is written for all processors.

Date: 01/9/23

* Parallel Programming

→ message passing parallel computers

→ send() / receive() } interprocess communication

→ a single program is written for all processors (Code Portability)

→ Message Passing Interface (MPI)

library of standard message passing routines

program starts with this → MPI_Init()

ends with this → MPI_Finalize() } setting up & tearing down the environment (creation/deletion)

MPI_Comm_size() → how many processors

MPI_Comm_rank() → rank of each processor

MPI_send() / MPI_receive() } for the actual message passing

- Single Program Multiple Data (SPMD)
  ↳ it is a style of programming

- Note: MPI is an interface, the implementation of each routine depends on the vendor

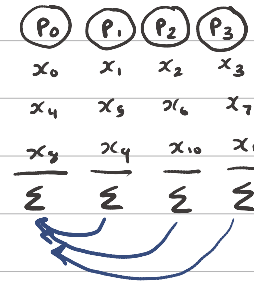
Date: /

Example Program: (running on n processors)

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

int main() {
  int i, x(i), tmp, sum=0,
      group-size, my-rank, N;

  MPI_Init();
  MPI_Comm_size(MPI_Comm_World,
                &group-size);
  MPI_Comm_rank(MPI_Comm_world, &my_rank);
```



Sum of numbers

Note: group-size = # processors
my-rank = assigned differently for each processor that runs this code

4/9/23

```
if (my_rank == 0)        → program flow for P0
{
  for (i=1; i < group_size; i++)
    MPI_send(&N, 1, MPI_INT, i, i, MPI_Comm_World);
```

start addr · # items · datatype · dest · tag

① Point-to-point communication
② Collective communication

Date: /

Assuming each processor already has access to all elements

```
for (i= my_rank ; i<N ; i= i + group_size)
    sum = sum + x(i);
for (i= 1 ; i< group_size ; i++)
 { MPI_ Recv ( &tmp, 1, MPI_Int i, i, MPI_Comm_World, &status);
                 receiving addr        same as send
    sum = sum + tmp;
    print (sum);
 }
} → END IF
else { → program Flow for P_i ∀ i≠0
  MPI_Recv(&N, 1, MPI_Int, 0, i, MPI_comm_World, &status);
                           src
  for (i= my_rank ; i<N ; i= i + group_size)
      sum= sum + x(i);

  MPI_send(&sum, 1, MPI_Int, 0, i, MPI_comm_World);
                             dst
}
MPI_Finalize()
} → PROGRAM END
```

Date: /

* For strings

MPI_Send (string, 10, MPI_CHAR, ...)

10 → length of string

MPI - Recv (string, 10, MPI_CHAR, ...)

MPI_CHAR → Important

→ to handle the heterogenity across diff architectures of storing different datatypes

Eg: P_0 → stores int as 2 Bytes

P_1 → stores int as 4 Bytes

→ also for underlying network

Processor X	Processor Y
Send (M, 64, Y, tag1)	Recv (P, X, 64, tag1)
Send (M, 32, Y, tag2)	Recv (Q, X, 32, tag 2)

* purpose of tags is to handle ambiguity at the receiver's end.

* we cannot guarantee in-order delivery

Date: 5/9/23

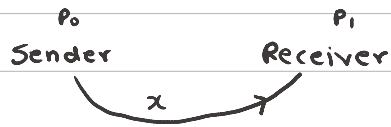
- In log P model, there is no explicit synchroniz--ation stage
  → But, pairwise synchronization happens during communication

Q. How do you send multiple non-consecutive elements using MPI_send()?

because send (start addr, #items, ...)

∴ sending A[0], A[5], A[10] ... together is an issue

Ans: copy the elements to a new buffer/array
⇒ it becomes consecutive ⇒ use MPI_send()



1) Synchronous_send() → waits till the elements are received before moving on.

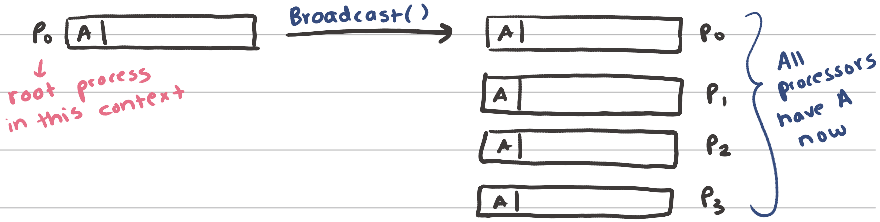
2) Asynchronous_send() → gives the elements to the output buffer & moves on

Sync	Async
→ reliability	→ no reliability assurance
→ more wait time	→ no wait time
→ doesn't req output buffer	→ output buffer is req.

Date: /

* Collective Communications

(1)



→ instead of using n-1 separate calls, data is sent to all processors in the comm_world.

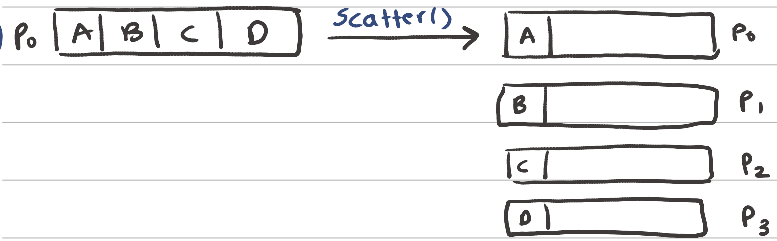
Adv

→ not just reduced # statements.

→ the vendor may optimize Broadcast() to $O(\log n)$ ≈ estimate

∴ It definitely has a speedup over n-1 send() statements.

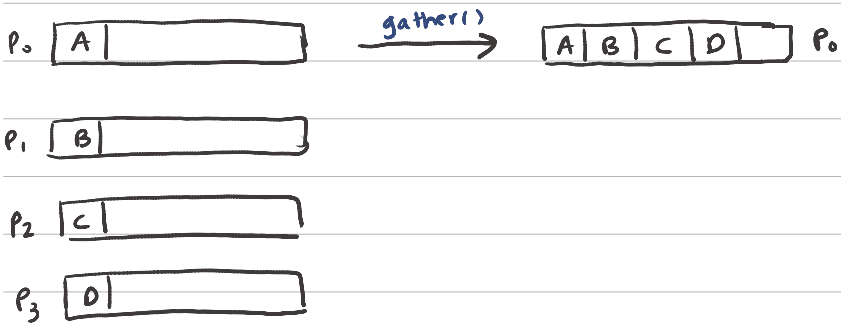
(2)



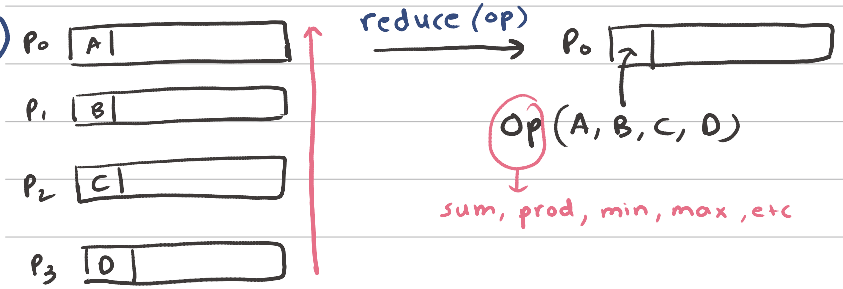
→ "personalized" communication

Date: /

→ the order is which the elements are scattered is determined by the ranks

③ 

→ collecting values from all processes into one.

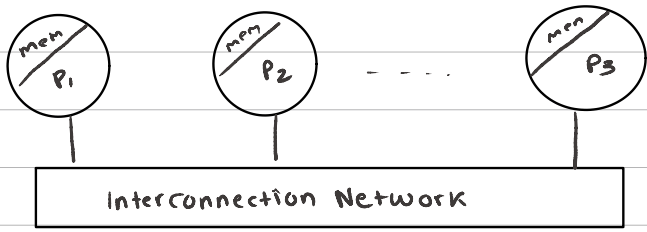
④ 

→ performs some sort of aggregation across all processors

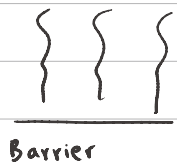
⑤ MPI_Barrier() → to make all processes in the MPI_world hit the synchronization barrier

Date: 11/9/23

* Message Passing Parallel Computer



MPI
- Create Terminate
- Who am I?
  - rank
  - size of communicator
- Function
  - point to point communication
  - collective communication

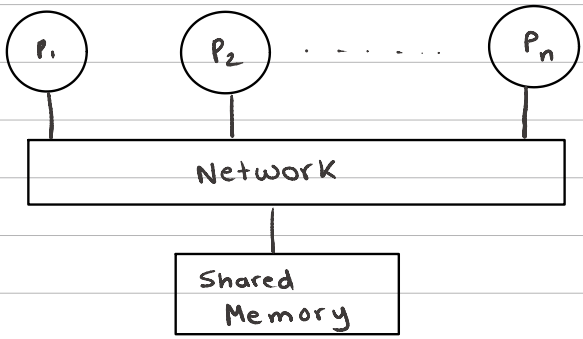


→ MPI_Barrier required for synchronization

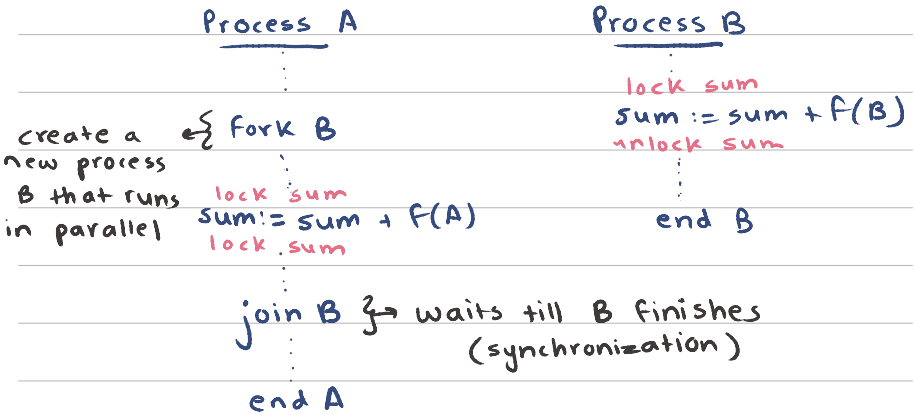
* In collective communication, the barrier is implied.
* In MPI, all functions are bi-directional

Date: /

* Shared Memory Parallel Computer



* each processor has its own local memory for doing computations (not shared)



* fork → create new parallel process
* join → synchronize with another parallel process

Date: /

Without lock

A	B
reads 0	sum = 0
computing	reads 0 (before A's write)
writes 0+1000 = 1000	computing
	writes 0+2000 = 2000

∴ lock is req.

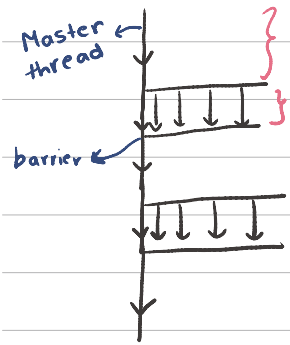
BUT, expected result is 3000

* Open MP - Standard → APIs for parallel machines with shared memory

Open specification for Multi-Processing

1. Compiler directives
2. Run-time library function
3. Environment variables

* Open MP talks in terms of threads, not processes.



* parallel threads share variables

* Only program is written for all threads.

* Thread creation is done using an OpenMP function call.

Date: /

* thread id = processor rank

* Code using OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
int noOfThreads, threadId, THREAD_COUNT=4;
#pragma omp parallel private (threadID)
                num_threads (THREAD_COUNT)
threadID = omp_get_thread_num();
printf ("I am thread %d \n", threadID);
#pragma omp barrier
if (threadID == 0)
{
noOfThreads = omp_get_num_threads()
printf ("Master: My team has %d threads\n", noOfTh);
}
return 0;
}
```

shared variable

fork = #pragma omp parallel private (threadID) num_threads (THREAD_COUNT)

* converting threadID to a put variable (specific to each thread)

run by all of the threads

destroys all threads

Date: /

* Output sample

I am thread 0
I am thread 1
I am thread 2
I am thread 3

→ can be printed in any order
- 4 threads are competing for std.io

Master: My team has 4 threads

* Master thread is included in THREAD_COUNT

12/9/23

omp_get_thread_num() → threadID

omp_get_num_threads() → no. of threads

#pragma omp parallel → creates llel threads

compiler directive

#pragma omp barrier → synchronization of threads

* Parallel Loop (code)

```
#include <stdio.h>
# include <omp.h>
# define N 7
int main (void)
{
```

Date: /

Parallel for loop in OpenMP

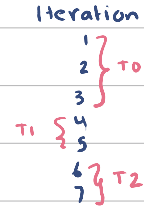
```
int i, THREAD_COUNT = 3;
#pragma omp parallel for num_thread (THREAD_COUNT)
{
    for (i=0; i<N; i++)
}
return 0;
}
```

→ execute the iterations of this for loop in parallel

Note:

* you dont have to explicitly create new threads when you use omp parallel for
* the for loop is unrolled & different iterations are assigned to THREAD_COUNT # threads.

Eg:



* the #iterations MUST BE KNOWN at compile time, so that it can be split across threads by the compiler.

Eg:

```
for (i= 2; i< 7; i++)
{
    a[i] = a[i] + c;
    b[i] = a[i-1] + b[i];
}
```

→ iterations are not independent.

∴ cant use parallel for

Date: /

* Cant use parallel for if there are dependencies across iterations.

dependency within iteration ✓

across iterations ✗

* you can provide additional information to the compiler via clauses

Eg: (i) Schedule (static, CHUNK_SIZE)

w/o specification = 1

used when iters take the same time

- Suppose chunk_size = 8, iterations 1 to 8 go to thread 1, 9 to 16 → thread 2 and so on (round-robin till all iterations are alloted to a thread)

(ii) Schedule (Dynamic, CHUNK_SIZE)

- if chunk_size = 8
  1) 8 iters are alloted to each thread
  2) if thread i completes its work first, then next 8 are given to i

(NOT ROUND ROBIN)

* different iterations take different times

(iii) Schedule (Guided, CHUNK_SIZE)

- Split the first $\frac{n}{2}$ iterations across all
- Then the next $\frac{n}{4}, \frac{n}{8}, \frac{n}{16}$

Date: /

* for scenarios where the workload decreases over the iterations

(iv) Schedule (auto, CHUNK-SIZE)

* when you don't know anything about the iterations.

* Data Parallelism - parallel for

* Functional

```
#pragma omp sections
{
    #pragma omp section
    { }
    #pragma omp section
    { }
}
```

* non-iterative
* creates threads
* each section is alloted to a thread & run in parallel

Eg: Sequential code

```
int A[max], i, sum = 0;
double average;
for (i=0; i< max; i++)
    sum = sum + A[i]
average = sum / max
```

Date: /

Parallel code

```
int A[max], i, sum = 0;
double average;
# pragma omp parallel for reduction (+:sum)
{
    for (i=0 ; i< MAX; i++)
        sum = sum + A[i];
}
average = sum / max
```

= aggregation

at the end, the partial sums computed by each thread is accumulated at the master thread.

- better alternative to locking & unlocking a shared variable "sum"
- here, we're just leaving it to the compiler to sum across all "sum"s and give it to the master thread.

Date: 15 / 9 / 23

* Synchronization : 1) # pragma omp critical
{ // block }

2) # pragma omp atomic
// one statement

3) # pragma omp barrier

4) # pragma omp parallel for nowait

- nowait ⇒ no barrier is added at the end
  ⇒ if we need a barrier, add it explicitly
- bydefault parallel for has a synchronization barrier at the end

5) # pragma omp flush [list]

omp_set_lock( )

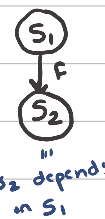
omp_ unset_lock( )

* Dependence Analysis

(a) Data dependence

S_1 : a = b × 2

S_2 : d = a + c



Flow dependence

RAW - Read After Write

Date: /

S_3 : $e = g \times 2$

S_4 : $g = f + h$

Anti-dependence

WAR - Write after Read

S_5 : $e = g \times 2$

S_6 : $e = f + g$

Output dependence

WAW - Write after write

(b) Control dependence

S_7 : if (a == 5) then

S_8 : b = 9

Eg: for i = 2 to 7 do

S_1 : a[i] = a[i] + c

S_2 : b[i] = a[i-1] + b[i]

endfor

* dependency across iterations
* compiler does thorough analysis + optimization to execute this loop in parallel.