S. No	Topic	Reference	Page/Section
1.	Introduction to parallel programming.	Introduction to Parallel Computing	1.1 to 1.2.4
2.	Programming models and parallel architectures, superscalar	Introduction to Parallel Computing	2.1
3.	Why pipelining? Performance analysis using pipelining on a single processor.	Introduction to Parallel Computing	2.1
4.	Impact of memory latency on performance. How to increase memory bandwidth.	Introduction to Parallel Computing	2.2, 2.2.1
5.	Cache and memory bandwidth. Performance considerations using the cache.	Introduction to Parallel Computing	2.2.2
6.	Shared and distributed memory architectures. Interconnection networks in distributed memory architectures.	Introduction to Parallel Computing	2.3, 2.3.1, 2.3.2, 2.4.2, 2.4.3, 2.4.4
7.	What is OpenMP? Basic program using OpenMP.	Introduction to Parallel Computing	7.10
8.	Programming using single thread.	Introduction to Parallel Computing	7.10.1
9.	Parallel programming using multiple threads.	IIT-Delhi Book	7.10.1
10.	Context switching threads, thread functions.	IIT-Delhi Book	
11.	Sequential consistency model	IIT-Delhi Book	
12.	Critical section, race conditions	IIT-Delhi Book	
13.	Handling of race conditions using openmp	IIT-Delhi Book	
14.	OpenMP scoping of variables	IIT-Delhi Book	
15.	Thread private variables	IIT-Delhi Book	
16.	Manual distributions of work and critical section	IIT-Delhi Book	
17.	Loops and reductions	IIT-Delhi Book	
18.	Vector-Vector operations, matrix-vector operations, matrix-matrix operations	IIT-Delhi Book	
19.	Tasks	IIT-Delhi Book	
20.	Variable accessing in task	IIT-Delhi Book	
21.	Completion of tasks and scoping variables in tasks	IIT-Delhi Book	
22.	Recursive task spawning	IIT-Delhi Book	
23.	Message passing interface	From class discussion	
24.	Basic MPI calls	From class discussion	
25.	Broadcast implementation using MPI	From class discussion	
26.	MPI Non blocking calls	From class discussion	
27.	MPI Collectives and MPI broadcast	From class discussion	
28.	Characterization of interconnects	From class discussion	
29.	Hockney model	From class discussion	
30.	Broadcast, Reduce, Scatter, Gather, Reduce Scatter, AlltoAll	From class discussion	
31.	Introduction to GPU architectures	Slides and class discussion	
32.	Data parallel algorithms, SIMD, SIMT,	Slides and class discussion	
33.	Introduction to CUDA programming	Slides and class discussion	
34.	Programme implementation using CUDA, Kernel launch	Slides and class discussion	
35.	Grid, blocks, CUDA functions	Slides and class discussion	

36. Multidimensional mapping of data space synchronization

37. Histogram

OpenMP

↳ multiProcessing

1. Hello world in ↳ multiple threads

```
#include<omp.h>
#include<stdio.h>
int main( int *argc, char *argv[]){
/* whenever we need a block to be
   executed in Parallel we have
   to enclose it in # Pragma omp
        compiler directive
  # Pragma omp parallel {
    Printf(" Hello world");
    }
  return 0;
}
```

header file ↳ containing prototypes for various OpenMP func

To compile & run →

```
$ gcc - fopenmp Helloworld.c
$ /a.out
```

To define no of threads

```
$export OMP_NUM_THREADS =4
$ ./a.out
```

env. variable

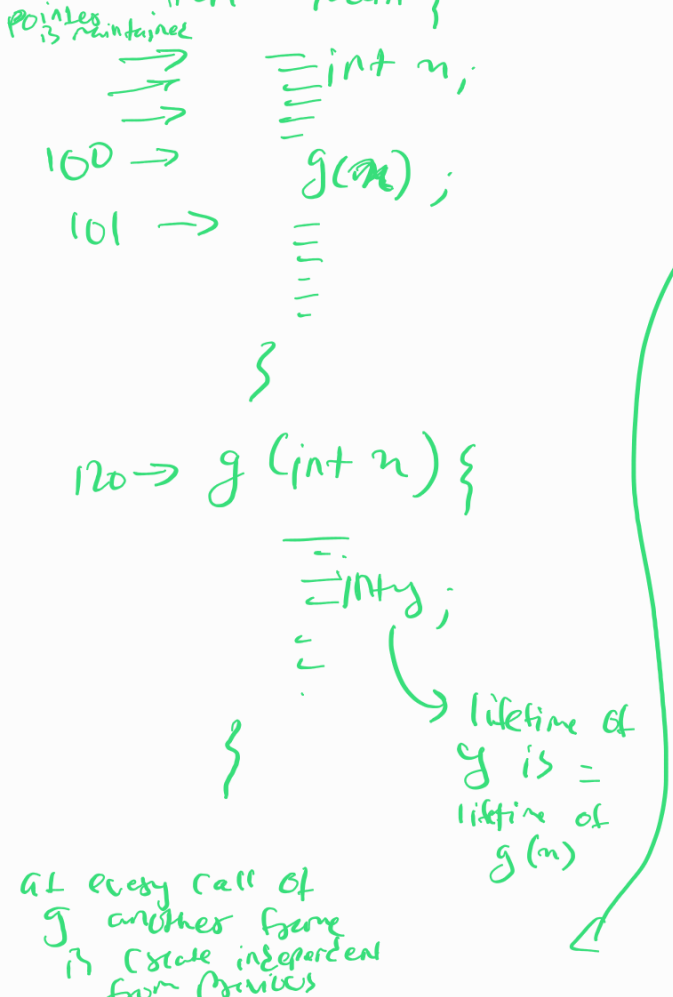
* Threads

↳ single stream of control in the flow of program

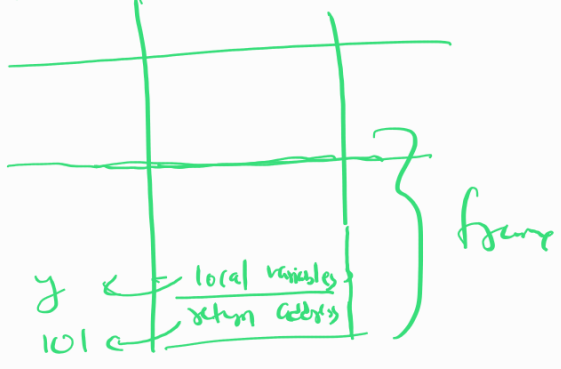
Note - Understanding memory of a program

```
int i = 4;
int j;

fun c                main {
                       int n;
100 →                  g(n);
101 →                  .....
                       }

120→ g (int n) {
       ....
       int y;
       ....
     }
```

lifetime of y is = lifetime of g(n)



At every call of g another frame is create independent from previous



Now for M/ Program

↳ we cant multiple strings of code flows

|||

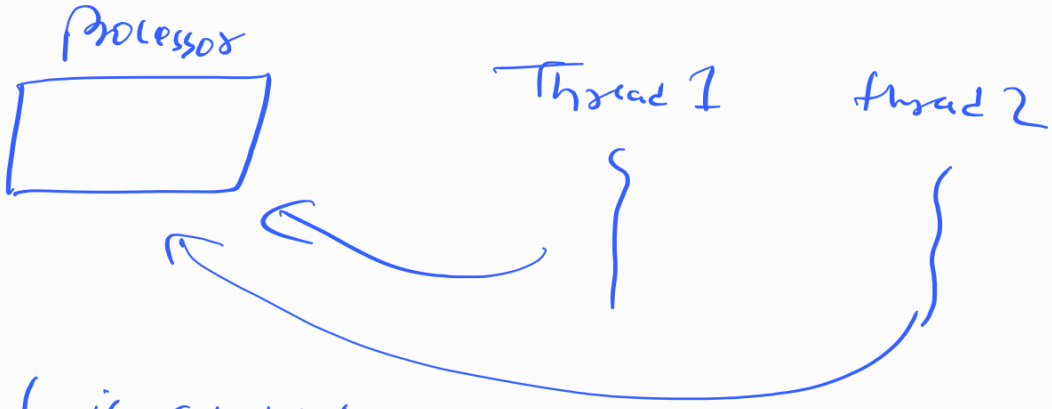
multiple threads

for doing so the main issue in memory of program will be stack as each thread will have it's own control of flows

↓

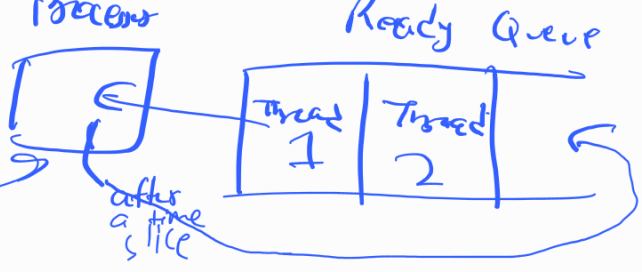
Here each thread has it's own stack

most devices are multitasking



thread 1 is scheduled for a time slice then thread 2, doesn't need to complete thread 1

↳ the scheduling is done by ready queue



Context Switching

↳ process by which an OS saves the current state ("context") of the executing thread or process & loads the previously saved state of another

↳ whenever a context is switch a copy is created of the registers and is stored in thread local space

On single core → Context switching creates an illusion of concurrency

On multi-core → true ||'ism is possible - multiple threads can run ||'ly each on its own core, context switching can still happen if needed

* OpenMP basic thread func^c

```
#include <omp.h>
#include <stdio.h>
int main (int *argc, char *argv[]) {
    #pragma omp parallel {
        /* get the total number of OMP threads
           int numt = omp_get_num_threads();

        /* get the thread id of this thread.
           int tid = omp_get_thread_num();

           Printf("Helloworld from thread %d of %d", tid,
                                                    numt);
    }
    return 0;
}
```

int numt, tid is inside the stack of a thread

Note- the order isn't sure at every run in the output, but execution takes place sequentially

to make it atomic we have to take help of locks & mutual exclusion.

note - if numt & tid is defined outside the parallel block.

```
int main (int *argc, char *argv[]) {
    int tid, numt;
#pragma omp parallel {
    numt = omp_get_num_threads();
    tid = omp_get_thread_num();
    Printf("Helloworld from thread %d of %d", tid, numt);
    }
    return 0;
}
```

output →

Hello world from thread 2 of 4

Hello world from thread 1 of 4

0 4 Refresh → 0 4

1 4 → 1 4

3 4 → 2 4

Race condition

Change in output thread 3 didn't print anything, what actually happened is thread 3's output was overwritten by thread

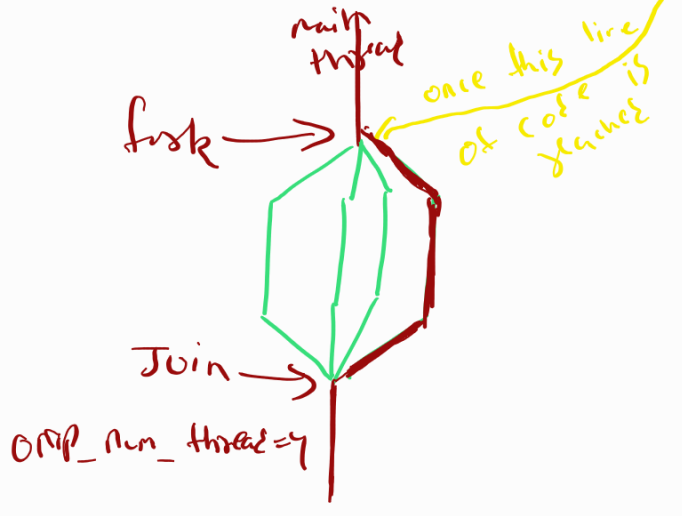
✳ About openMP

- API for shared memory model programming
  - → compiler derivative (e.g- # Pragma omp parallel)
  - → Runtime Library routines (e.g- omp_get_num_threads)
  - → Env variables (e.g- OMP_NUM_THREADS)

- Adv -
  - → Standardization & Portability
  - → Ease of use

- Thread Model → threads share memory space except stacks

- Fork-Join model
  - → # Pragma omp parallel {

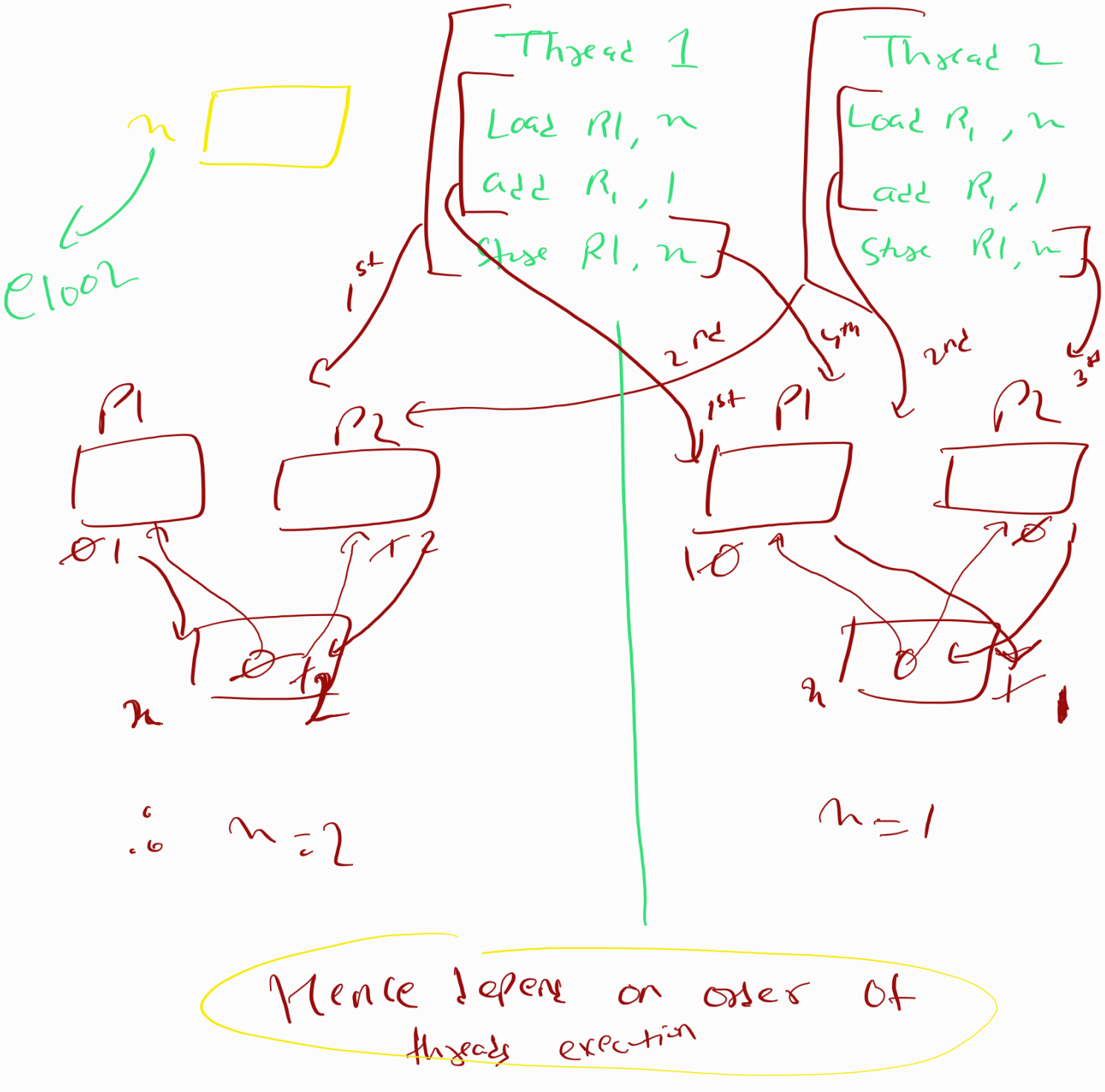


}

* Race condition -

If correctness of the ∥ program depends on the order in which threads are scheduled.

e.g



To fix Race condition -

e.g-

```
int main ( - - - ) {
    int numt, tid;
    # Pragma omp Parallel {
        numt = omp_get_num_threads();
        tid = omp_get_thread_num();
        Prit( Hello - - - - - , tid, numt);
    }
    return 0;
}
```

```
int main( - - - - ) {
    int numt, tid;
    /* Specify Private & Shared variable
    # Pragma omp parallel Private(tid) Shared (numt)
        numt = - - - - - ,
        tid = - - - ;
        Printf( - - - , tid, numt);
    }
    return 0;
}
```

means every thread has its own tid

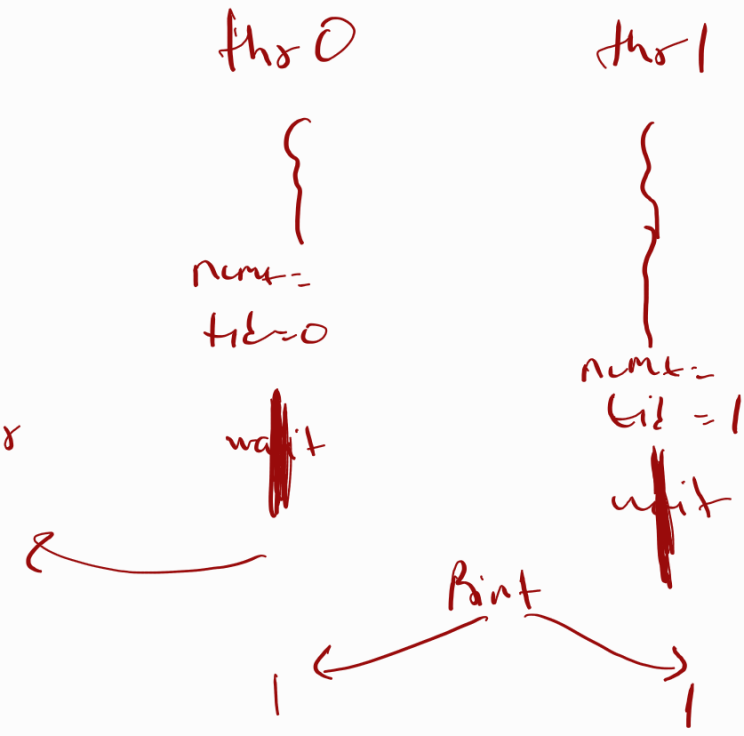
variable is global & everyone access same location

Note- default behaviour for variable in openmp is Shared.

Note — Still how can we make sure that the bug is fixed for sure.

By making the code with strace fail (almost everytime).

```
int main (- - -.) {
  int numt, tid;
  # pragma omp parallel {
     numt = omp_get_num_threads();
     tid = omp_get_thread_num();
  /* introduce wait so other thread get scheduled.
     for (j=0 -> 100000000; j++);

     Printf(Hello - - - -., tid, numt);
  }
return 0;
}
```



but adv. compiler might fix this

so now fix fixing + fix size check

```
int main (_ _ _.) {
    int numt, tid;
  # pragma omp parallel private(tid) shared(numt){
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

    /* introduce wait so other thread get scheduled.
       for (j=0 -> 100000000; j++);
      Pith( Hello - _ _ _ _ , tid, numt);
    }
  return 0;
}
```

now as we can observe that num_t is a shared resource hence, it's getting called multiple times, so let's try to define it globally

```
int main (_ _ _.) {
   int tid;
   int numt = omp_get_num_threads();
# pragma omp parallel private(tid)

   tid = omp_get_thread_num();


   Printf("Hello - - - - -", tid, numt);
 }
return 0;
}
```

output →

Hell - - - - thread 2 of 4
3 of 4
0 of 4
1 of 4

bcz when defined globally the no of thread was only 1

now to avoid calling numt multiple times and at the same time not defining it globally below is the soln —

↳ only one thread can call numt

```
int main (- - - -) {
  int tid; numt

# Pragma omp Parallel default (shared)

  tid = omp_get_thread_num();
  /* Initialise numt if tid==0
    if (tid==0) {
      numt = omp_get_num_threads();
    }
  Printf( Hello - - - - , tid, numt);
}
return 0;
}
```

output →

Hello world from thread 0 of 4 - - - 0 of garbage value

1 of 4 run again - - - - 1 of -2.31

2 of 4 - - - - 2 of 4

3 of 4 - - - - 0 of 4

to make it fail everytime (almost)

```
int main (_ _ _.) {
   int tid; numt

#Pragma omp Parallel default (Shared)

    tid = omp_get_thread_num();

    /* Initialise numt if tid==0
      if (tid==0) {
          /* introduce wait so other threads get schedule
          for (j=0 -> 100000; j++) {

        numt = omp_get_num_threads();
      } }

   Pritf(Hello - _ _ _ _, tid, numt);

 }
return 0;
}
```

so now for fixing + use check

```
int main(- - - - ) {
    int numt;
#pragma omp parallel default (shared) {
        tid = omp_get_thread_num;
        if (tid == 0) {
            for (j=0 -> 1000000; j++) {
            numt = omp_get_num_threads();
                }
        }
# pragma omp parallel default (shared) {
        int tid = omp_get_thread_numbes;
        Printf (" Hello - - - -          )
        }
    return 0;
}
```

→ not elegant

↓

bcz calls useless

able to reduce calls.

Another way to reduce multiple calls

→ Thread Private variable

→ making tid persistent across || sections.

directive used to give thread it's own instance of a global variable & keep that instance persistent across multiple || region

Middle ground b/w shared & private

- Each thread has it's own copy of variable
- But the value persist across different parallel sections unlike private where it gets reinitialised

```
int tid;
# Pragma omp threadPrivate (tid)
int main( int *argc, char *argv[]){
    int num;
    # Pragma omp Parallel default (shared){
        tid= OMP_get_thread_num();
        if (tid==0){
            for (j=0→100000 ){
                num+= OMP_get_num_thread;
            }
        }
    # Pragma omp parallel default (shared){
        Printf (" Hello world %d %d", num, tid)
        }.
    } //main 0;
```

Note- Thread Private only applies to file scope / static variables

Note- we usually avoid using implicit barrier using # Pragma omp Parallel as barrier

Here, there's a compiler directive for same use

```
int tid;
# Pragma omp threadPrivate (tid)
int main( int *argc, char *argv[]){
    int num;
    # Pragma omp Parallel default (shared){
        tid= OMP_get_thread_num();
        if (tid==0){
            for (j=0→100000 ){
                num+= OMP_get_num_thread;

    # pragma omp barrier

    Printf (" Hello world %d %d", num, tid)
    } //main 0;
```

if we want only one thread to perform any activity in a syncronised manner

↙

Pragma omp single

```
int main (int* argc, int* argv[]) {
    int numt;
    # Pragma omp parallel default(shared){
        int j, tid;
        # Pragma omp single {
            for(j=0 → 10000000) {
            numt = omp_get_num_threads();
            }
        tid = omp_get_thread_num();
        Printf("Hw %d of %d", tid, numt);
    }
```

only one thread executes this comman & at the end there's synchronisation

if

other threads can go forward

```
# Pragma omp single nowait {
    for(j=0 → 10000000) {
    numt = omp_get_num_threads();
    }
```

if

only master thread is executed ↓ tid=0

```
# Pragma omp single master{
    for(j=0 → 10000000) {
    numt = omp_get_num_threads();
    }
```

* Shared memory consistency models

memory consistency rules → set of rules which defines the order in which reads & writes

hardware & compiler might reorder the code for optimization

Contract b/w Programmers & system

1. Sequential consistency model -

→ operation of all processor appears in some global order that is consistent with each processor's local order of operation.

→ very simple to reason

→ Expensive to implement

→ processor & compiler can't reorder memory ops. freely.

2. TSO (Total store order)

3. release consistency

4. weak consistency model

note —

omp_get_wtime() = time in second

omp_get_wtick() = resolution of timer

I. Computing Sum

Sequential code

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

#define ARR_SIZE=
        1000000000
int a[ARRSIZE];

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

    sum = 0;
  double t1, t2;
  for(int i=0 -> ARRSIZE){
        arr[i] = 1;
  }
  t1 = omp_get_wtime();
  for(i=0 ->ARR_SIZE){

        sum += arr[i];

  }
  t2 = omp_get_wtime();
printf("sum = %d: time = %g", sum, t2-t1)
 return 0;
}
```

Parallel code -

asking each thread to do same task

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

#define ARR_SIZE=
        1000000000
int a[ARR_SIZE];

int main(int *argc, char *argv[]){
  int sum=0; int numt, tid;
    double t1, t2;
  for(i=0 -> ARR_SIZE) {
        a[i]=1;
      }
  t1 = omp_get_wtime();

#pragma omp parallel private(tid){
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();
      for(int i=0 -> ARR_SIZE) {
            sum+=a[i];
            }
      }

  t2 = omp_get_wtime();

  printf("sum=%d. time=%lf", sum, t2-t1);
return 0;
}
```

Problems -

- Incorrect — not truly Parallel
- race condition in sum
  - sum > 1 billion
  - $t_p > t_s$ → cache coherency
  - expected sum = 4 billion → 4 x 1 billion (4 = default no of threads)
- sum is shared variable

Output →

sum = 137 361697. time = 3.1103

fixing // code

```
int tid, numt, sum=0;
double t1, t2;
for(i=0 -> ARR_MAX) {
    arr[i]=1;
}
t1 = omp_get_wtime;
#pragma omp parallel private(tid) {
    int from, to;
    tid = omp_get_thread_num();
    numt = omp_get_num_threads();
    from = (ARR_SIZE / numt) * tid;
    to = (ARR_SIZE/numt) * (tid+1) -1;
    if (tid == numt-1) {
        to = ARR_SIZE -1;
    }
    for (i = from -> to) {
        #pragma omp critical{
            sum += a[i]
        }
    }
}
with( . . . . )
```

- Incremental problem solved now distribution of work;
- $t_p < t_s$

very important to ensure section is atomic so at one time only one thread can access.

→ critical section

a lock is maintained to do so

lock itself run into a race condition sometime

left the & updated value to 1

so test & set(x) is used

even after # pragma omp critical the time will not significantly hence we will introduce Psum (local variable)

```
# include<omp.h>
# include <stdio.h>
# Define ARR_SIZE = 100000000;
int a[ARR_SIZE];
int main(int * argc; char *argv[]) {
    int tid, numt, sum=0;
for( int i = 0 → ARR_SIZE) {
        a[i] = 1;
    }
# pragma omp parallel default(shared) Private(tid){
    int Psum=0, from, to;
    tid = OMP_get_thread_numbes();
    numt = OMP_get_num_threads();

    from = (ARR_SIZE/numt) * tid
    to = (ARR_SIZE/numt) * (tid+1) -1;
    for (i= from → to) {
        Psum += a[i];
    }
# pragma omp critical {
    Sum += Psum;
}
```

```
# include(stdio.h)
# include(omp.h>
# Define ARR_SIZE = 10000000
int a[ARR_SIZE];
int main(int *argc, char *argv[]){
    int      , sum=0;
    for (int i=0 -> ARR_SIZE){
        a[i]=1;
        }
    # Pragma omp Parallel default(shared) {
        int Psum=0;
        # Pragma omp for {
        for(i=0-> ARR_SIZE) {
            Psum+=a[i];
            }
        # Pragma omp critical {
            sum+= Psum;
            }
```

↓ to reduce the effect of Psum

```
# Pragma omp Parallel default(shared) Reduction(+:sum)
    # Pragma omp for {
        for(i=0-> ARRSIZE){
            sum+=a[i];
            }
    }
```

gives each thread it's private copy of sum

How to even slice the pragma omp for

```
# pragma omp parallel for default(shared)
reduction(+:sum){
for(i=0 -> ARR_SIZE){
  sum += a[i];
}
```

* Linear Algebra

BLAS (Basic Linear Algebra subroutines)

i) BLAS-1 Vector, vector-vector
ii) BLAS-2 Vector-matrix
iii) BLAS-3 matrix-matrix

† Dot Product ($y = A \cdot B$)

Sequential →

```
int N;
dot = 0
for (i=0 -> N){
  dot += A[i] * B[i];
}
```

Parallel →

```
int N;
dot = 0;
# pragma omp parallel for
schedule(static/dynamic, chunk_size){
for (i=0 -> N){
  dot += A[i] * B[i];
}
```

Fixed value at a time

slice up N iterations & have them to threads in team

equally sized blocks

II. Matrix X Vector

sequential ->

```
for (i=0 -> n) {
  for (j=0 -> n) {
    y[i] = A[i,j] * x[j];
  }
}
```

||->

Note- C stores matrix in Row major order

↓

Hence in C column is not contiguous

||

false sharing situation

((cache line size is too large, even after not sharing it creates illusion like sharing))

↓

Sol^ -> store in Private variables

```
#pragma omp parallel for schedule(dynamic, chunk size)
for (i=0 -> n) {
  for (j=0 -> n) {
    y[i] += A[i,j] * x[j];
  }
}
```

II Matrix - matrix multiply

the column major in C won't be much of an issue but the column major in B is an issue and here we can't get benifit of data locality

Theoretical

$$\frac{\#\text{ computations}}{\#\text{ memory ops}} = \frac{2n^3}{3n^2} = O(n)$$

we want to be compute bound

so what we can do is divide the matrices in $b \times b$ blocks

$$\frac{\#\text{ computations}}{\#\text{ memory}} \Rightarrow$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} + \dots$$

* Tasks

→ Piece of code that needs to be executed at some point of time.

e.g. - counting array sum using task

```
sum=0
ARR_SIZE 600
STEP_SIZE 100
int a[ARR_SIZE];

#pragma omp parallel {
    #pragma omp for {                    → i+=STEPSIZE
        for (i=0 -> ARR_SIZE){
        start=i, end = i+ step_size-1;
        printf( - - - - )

    #pragma omp task {
        int psum=0
          printf( - - - - )
          for ( i=start -> end ){
              psum+=a[i];
          }
    #pragma omp critical {
        sum+=psum;
    }
}
printf( . . . )
```

Task Queue

Executed using Linked List

```
struct node {
    int data;
    node * next;
}

# Pragma omp parallel {
    # Pragma omp single {
        for (Ptr = head; Ptr != NULL ; Ptr = Ptr->next)
            # Pragma omp task {
                complete_inverse (Ptr->retorit);
            }
    }
}
```

a single thread out of the forked thread performs calling

Note - To access visible data task Here is data env. associated to task there copy of all variables of thread of store.

Note - #pragma omp parallel firstprivate( )

create copy of variable & initiate the variable just before this point

Note - #pragma omp for nowait( )

generally, most of the for has an implicit barriers at the end of their code, nowait will & tells that barrier & start executing next lines

Note - #pragma omp barrier

for directives which doesn't have a implicit barrier or have nowait - to put a barrier barriers.

Note - #pragma omp taskwait

wait by all tasks spawned by this task to complete

it's not a barrier

Using task for recursive splitting

```
int do_sum(int *start, int end){
    int mid, x, y, res;
    if (end == *start) {
        res = a[start];
    } else {
        mid = (end + start)/2;
        printf("[TID: %d] sum(%d, %d) = sum(%d, %d) + sum(%d, %d),
            tid, start, end, start, mid, mid+1, end);
#pragma omp task shared(x)
        x = do_sum(start, mid);
#pragma omp task shared(y)
        y = do_sum(mid+1, end);
#pragma omp taskwait
        res = x + y;
    }
```

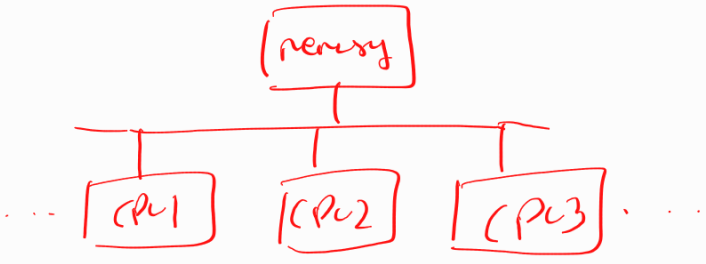
→ not a busy wait, just wait for tasks spawned by task to complete

```
#define ARR_SIZE 3
#define STEP_SIZE 1
int a[ARR_SIZE];
int main(int *argc, int *argv[]){
    int i, sum = 0;
    for(i=0 -> ARR_SIZE){
        a[i] = 1;
#pragma omp parallel
#pragma omp single
    sum = do_sum(0, ARR_SIZE-1);
    printf("sum = %d", sum);
    return 0;
}
```

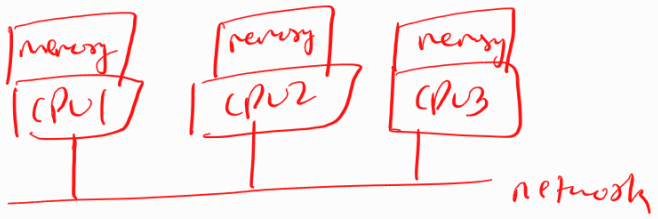
Open MPI

Parallel Architecture & Programming models

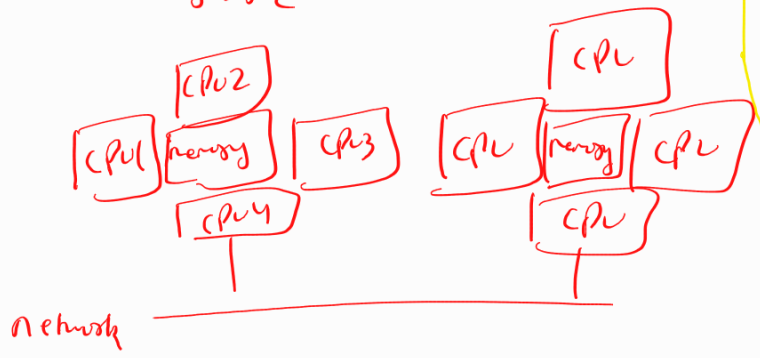
- Shared memory



- Distributed memory

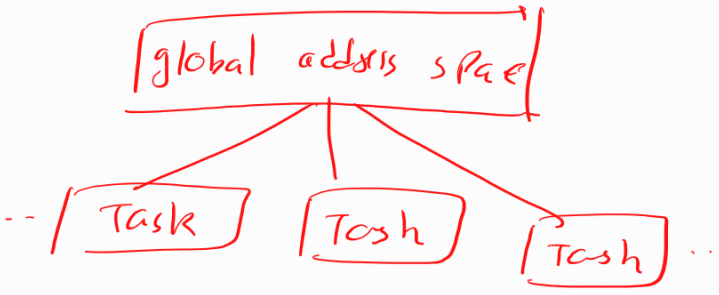


- Hybrid



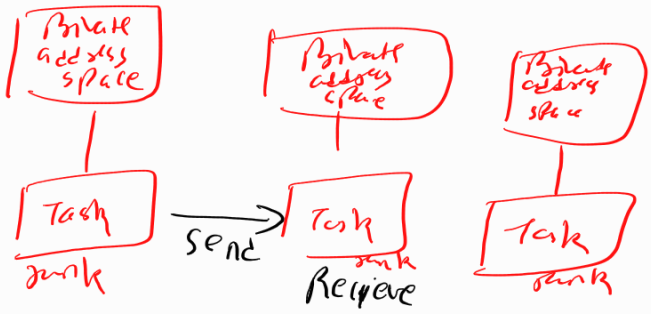
- Shared memory

OpenMP



- Message Passing

MPI



- message Passing +

OpenMP + MPI

MPI - Message Passing Interface

↓

Specification of a library used by message Passing Parallel Programming model in Distributed memory System

Compile → mpi compiler → mpicc C — C
mpic++ → C++
mpixx → C++

to run → [mpirun / mpiexec - np 4 a.out]
np → no of processor
4 Processor
goes to a scheduler

Note — rank ← [0, n-1]

Structure of MPI code

```
#include "mpi.h"
:
main(...) {
  :
  MPI_Init(....);
  :
  work ⟶ using MPI calls for message passing
  :
  MPI_Finalize(....);
}
```

Note — MPI → all process will execute same code, if you want one process to do something diff. then we take use of rank to differentiate

Few important MPI calls → like, MPI_COMM_WORLD → communicator

1. `MPI_Comm_size(comm, &size);`
   → returns total no of MPI process

2. `MPI_Comm_rank(comm, &rank);`
   → unique identifier to process

- Implementing Broadcast

→ Some rank has data and it wants to pass that data to other ranks

```
#include "mpi.h"
:
main(...) {
    int bvalue, numranks, rank;
    MPI_Init(...);
    MPI_Comm_size(MPI_COMM_WORLD, &numrank);
    MPI_Comm_rank(MPI_COMM_WORLD, rank);
    if (rank == 0) {
        bvalue = 100;
    }
    if (rank == 0) {
        MPI_Send(&bvalue, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
    }
    else if (rank < numrank-1) {
        MPI_Recieve(&bvalue, 1, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
        MPI_Send(&bvalue, 1, MPI_INT, rank+1, 1, MPI_COMM_WORLD);
    }
    else {
        MPI_Recieve(&bvalue, 1, MPI_INT, rank-1, 1, MPI_COMM_WORLD, &status);
    }
```

To Broadcast—

0 → 1 → 2 → 3 ---- numranks

1

MPI Send {
Pointer to buffer,
1 = count,
Element type,
Destination,
1 = tag,
COMM = communicator,
}

MPI Recieve {
Pointer to buffer,
1 = count,
Element type,
Source,
1 = tag,
COMM = communicator,
& Status,
}

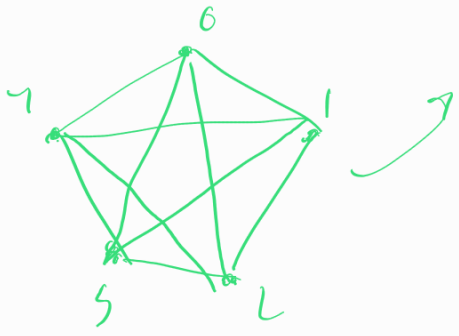
note - Blocking calls - doesn't return until the MPI library has completed the necessary work
e.g. MPI_Send(...), MPI_Recv(...)

Non-Block calls - returns immediately, handing you a MPI_Request while the actual data transfer proceeds in background
e.g. MPI_ISend(...), MPI_IRecv(...)

if we think this way of broadcasting might be slow, if per send call cost = t then, (n-1)t cost

but can we use a better strategy for more efficient calls making

i) Clique (complete graph):



many calls has to be made from each node → not scalable

ii) Mirroring type (Recursive Doubling):

1st : 0 → 1
2nd : 0 → 2, 1 → 3
3rd : 0 → 4, 1 → 5, 2 → 6, 3 → 7

} exponentially will keep track of doubling behaviour → 2^i iteration on i

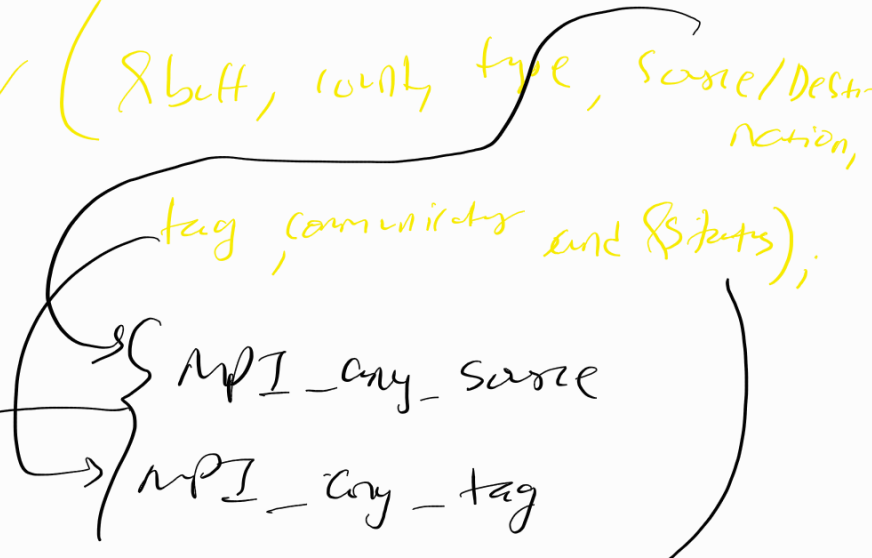
Sending data
1st : 0
2nd : 0, 1
3rd : 0, 1, 2, 3

Receiving
1st : 1
2nd : 2, 3
3rd : 4, 5, 6, 7

Algo to implement it

```
for (int ctr=1; ctr<numrank; ctr *=2) {
    if(rank < ctr) { /* i have data */
        if (rank + ctr < numrank)
    MPI_Send(&value,..., rank+ctr, ...);

    else(rank < 2*ctr) {
    MPI_Recv(&value,..., rank-ctr, ...);
}
```

Note - MPI_Send/Recv (&buff, count, type, source/Destination, tag, communicator and &status);



To identify which

Status.MPI_source

Status.MPI_tag

MPI_Status

↓

MPI_Get_Status(&status, datatype, &count)

- MPI collectives
  - operations that are performed by all ranks together
    - i) synchronisation operation
    - ii) Data movement (Distribution & Collection)
    - iii) collective computation → like Reduce operation

1. Synchronisation -

a) MPI_Barrier(comm) → MPI_COMM_WORLD

```
main( ) {
    ⋮
    compute
    MPI_Barrier( ... );   → early reached Boxes has to wait for all the Boxes to complete.
    ⋮
}
```

3) MPI_Bcast(&buffer, count, datatype, root, communicator);

diff for diff calls

Broadcast

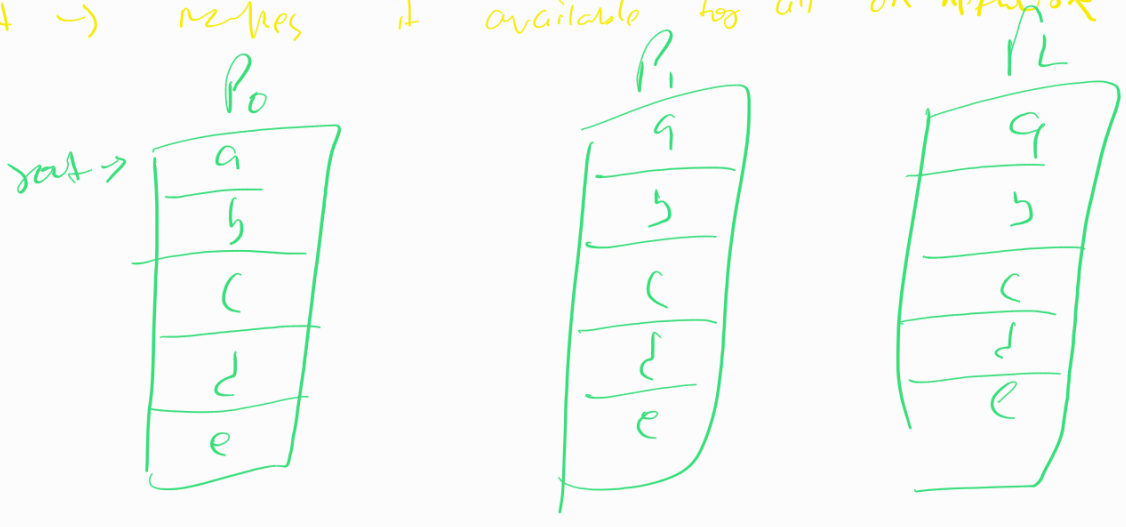
points to the start of data

rank which is doing the Bcast

Note- MPI_Bcast is more efficient than MPI_Send

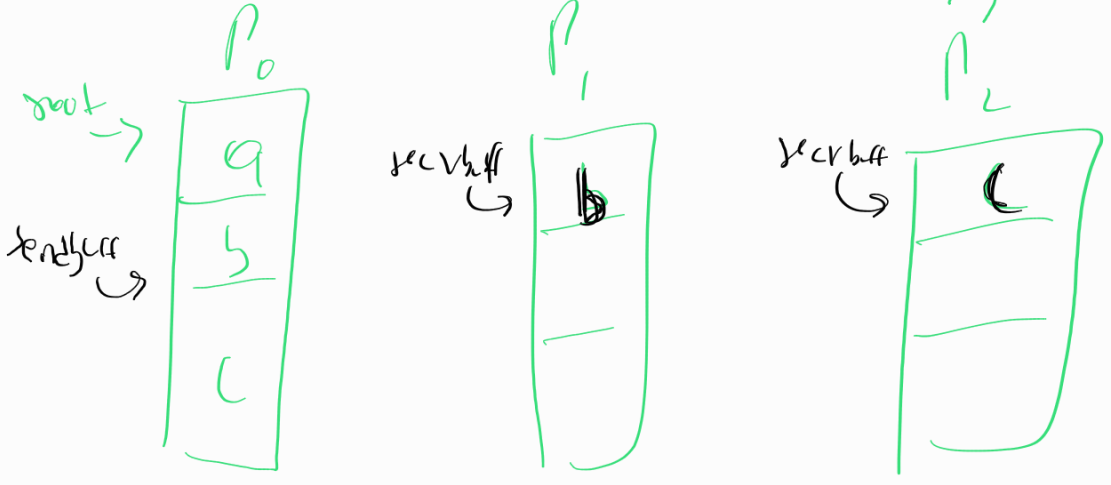
Send → transfers data node to Node

Bcast → makes it available for all on network



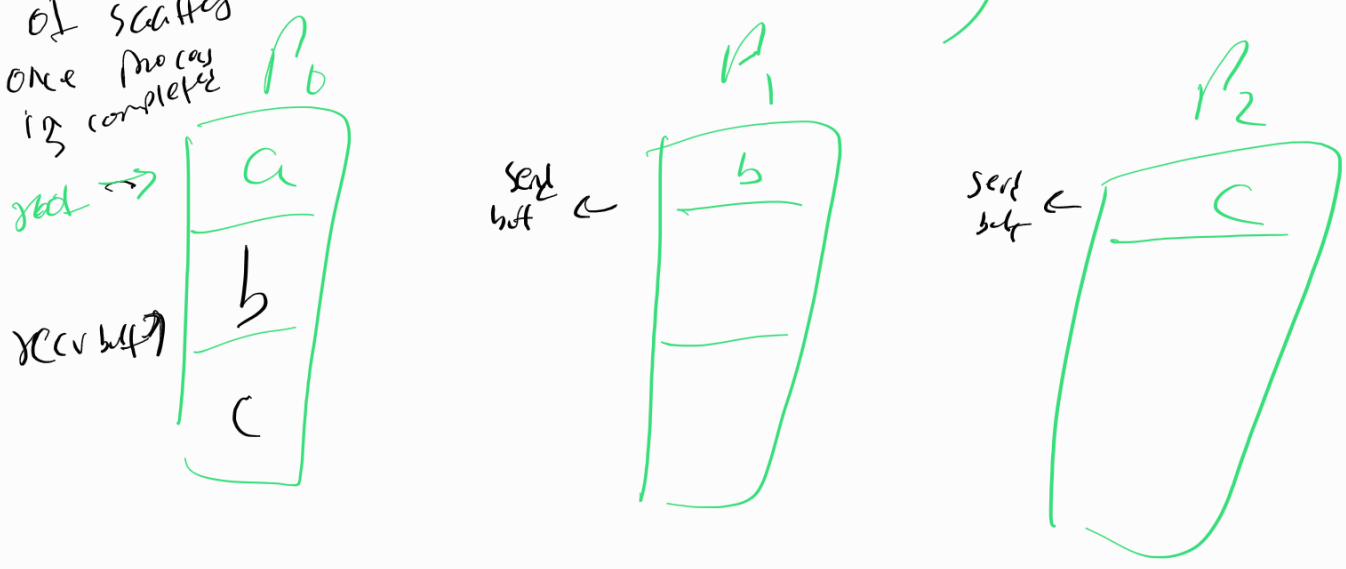
2. Data Movement → in a master-slave type setting Scatter helps in distributing data to other processors.

a) MPI_Scatter(&sendbuff, sendcnt, sendtype, &recvbuff, recvcnt, recvtype, root, comm);



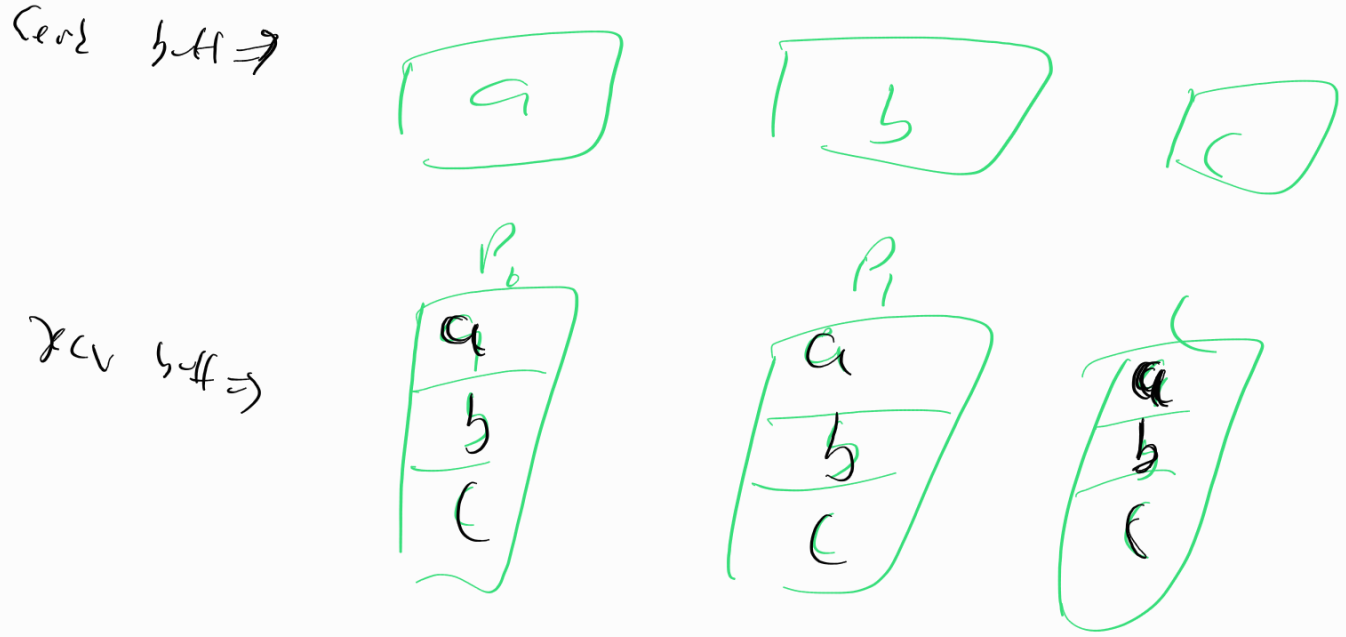
b) MPI_Gather (_ _ _
_ _ _ _
_ _)

reverse work of scatter once process is complete



c) MPI_Allgather(&sendbuff, sendcnt, sendtag,
&recvbuff, recvcnt, recvtag,
communicator)

no root



note- Blocking calls → buffer isn't reusable

- Non-Blocking calls → to overlap communication with computation

a) MPI_ISend(....)

↳ returns immediately, even if data hasn't been fully copied out

↳ buffer becomes not usable till it has completely been copied

MPI_ISend(&buf, cnt, datatype, destination, tag, comm, &request)

↳ MPI_Request

b) MPI_IRecv(...)

↳ when we don't know when we'll be recieving message back

MPI_IRecv(&buf, cnt, datatype, source, tag, comm, &request)

Note -

MPI_Test(&request, &flag, &status)

if completed => flag = 1
if not completed => flag = 0

int

MPI_Status

help us know the progress of Isend

MPI_TestAny(count, &array of request, &index, &flag, &status);

Checks if any of the request is completed

MPI_TestAll

Note - MPI_wait(&request, &status)

A Blocking call which waits for all non-blocking calls to be completed.

MPI_waitall(count, &array of request, &array_of_status);

MPI_waitAny

- Application : Distributed Histogram updation
  → Large histogram

3 Collective computation

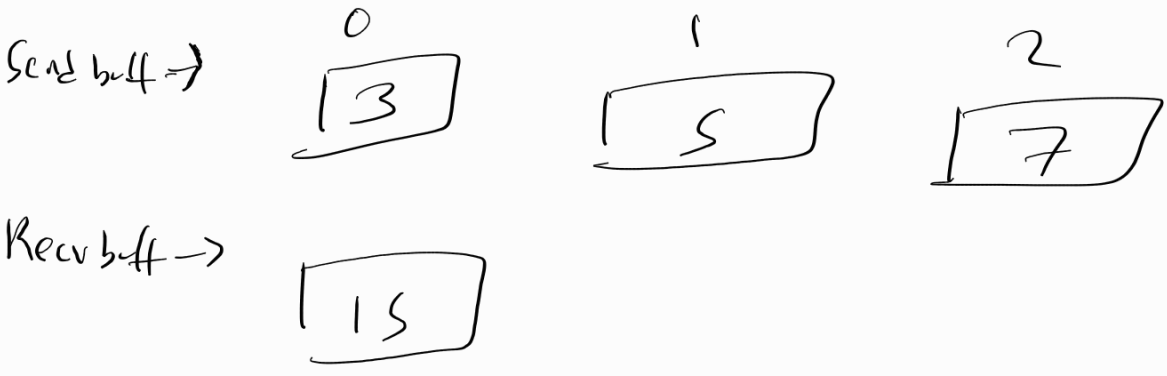
a) MPI_Reduce(& sendbuff, & Recvbuff, cnt, datatype, op, root, comm);

op → Operation to be performed

root → the rank in which result is expected

MPI_SUM
MPI_MAX
MPI_MIN
⋮

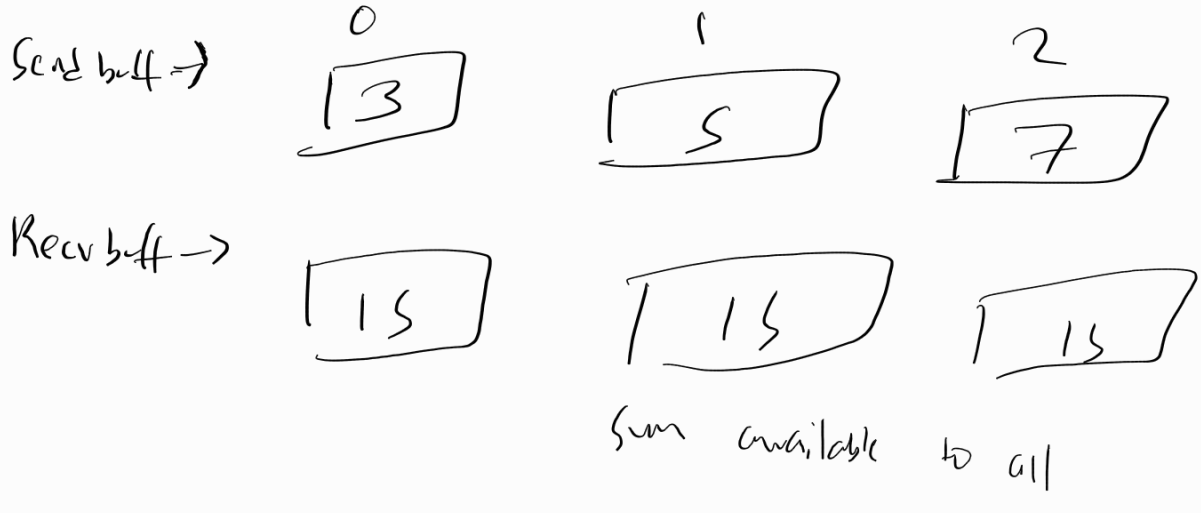
e.g. MPI_sum



MPI_AllReduce(- - - - - - - -, root, - -)

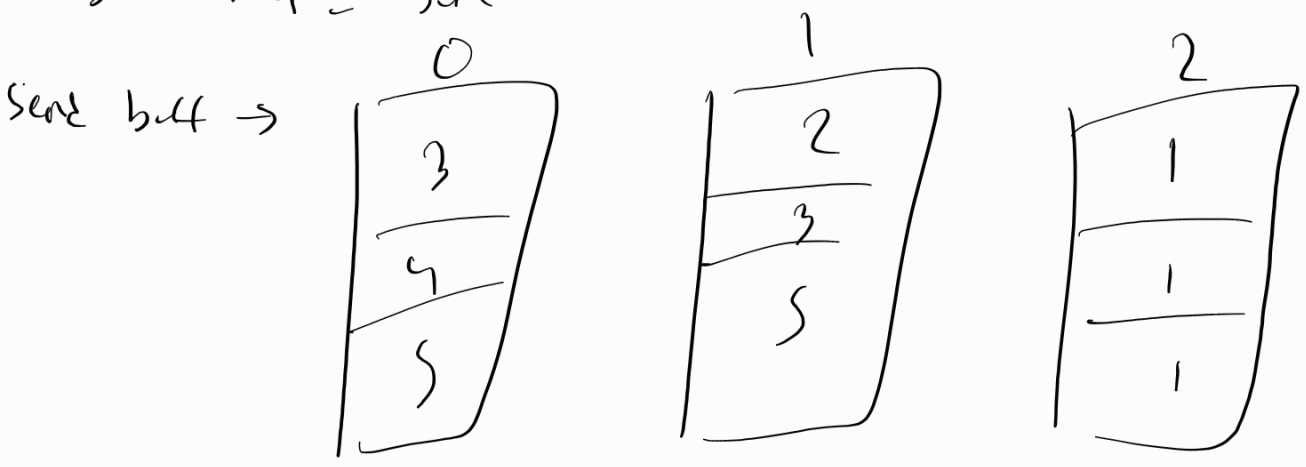
root → no root

e.g. MPI_sum

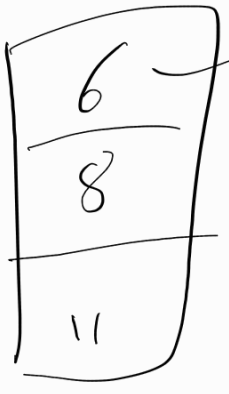


5) MPI_Reduce_Scatter(&sendbuf, &recvbuf, recvcount, datatype, op, communicator);

eg MPI_Sum

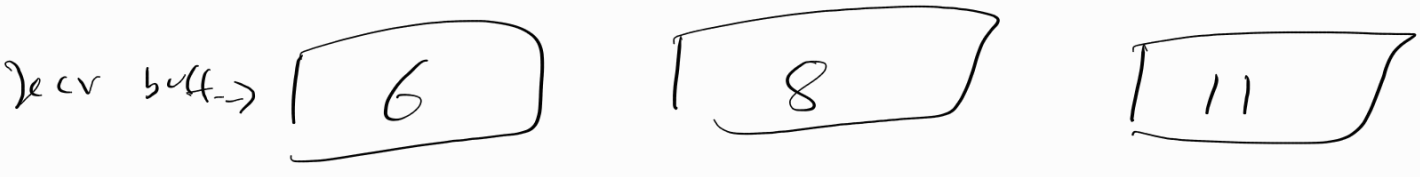


Stn-1 ⇒ reduce ⇒

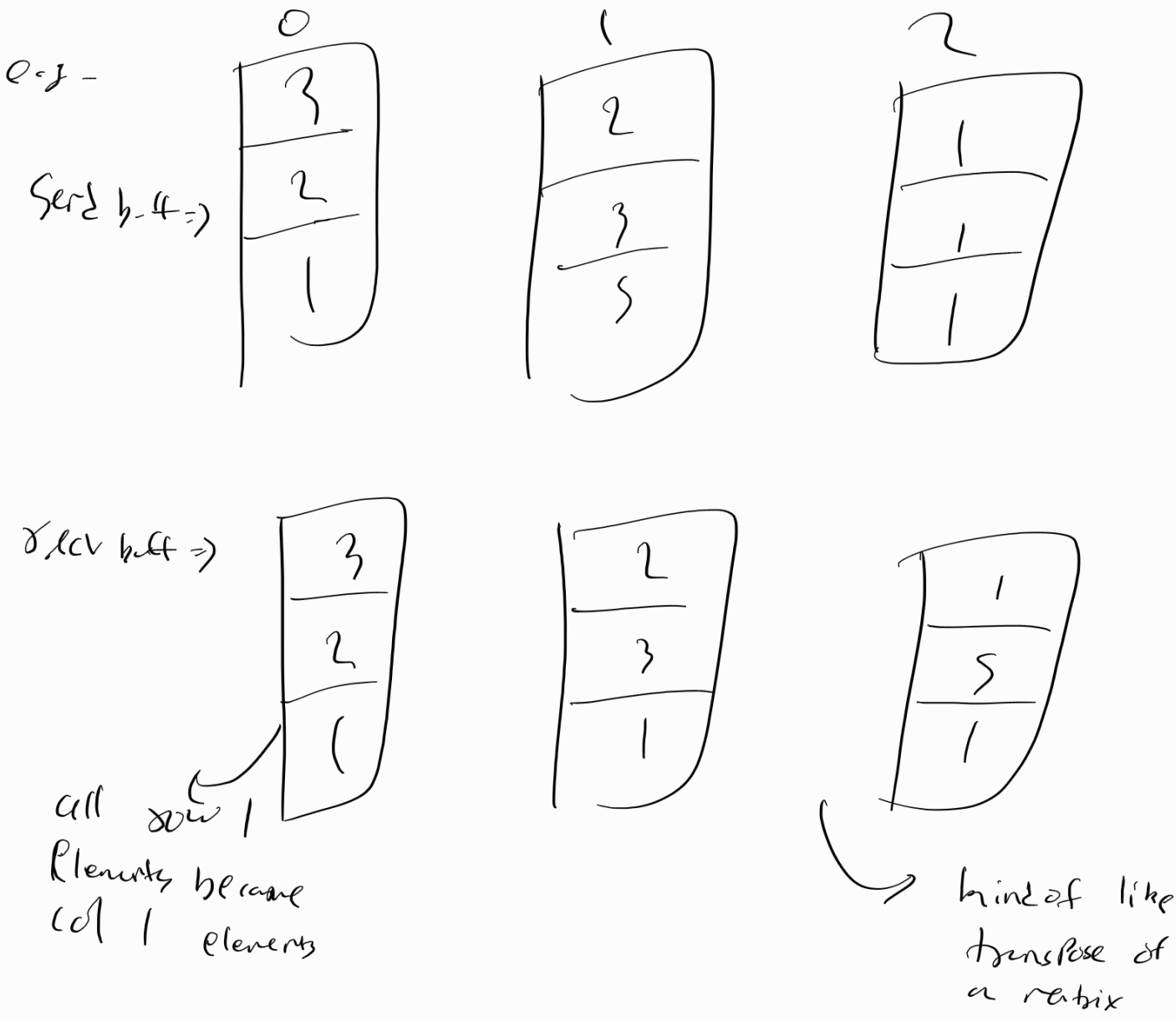


Sum of first two elements

Stn-2 ⇒ Scatter ⇒



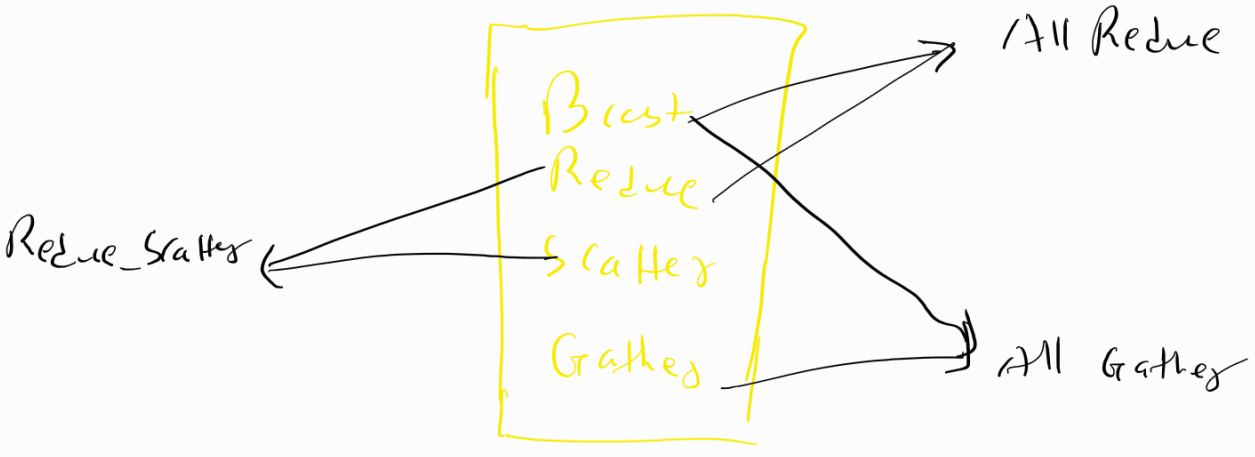
() MPI_ All to All (& Sendbuff, sendcount, sendtype
& recv buff, recv count, recv type,
(comm);



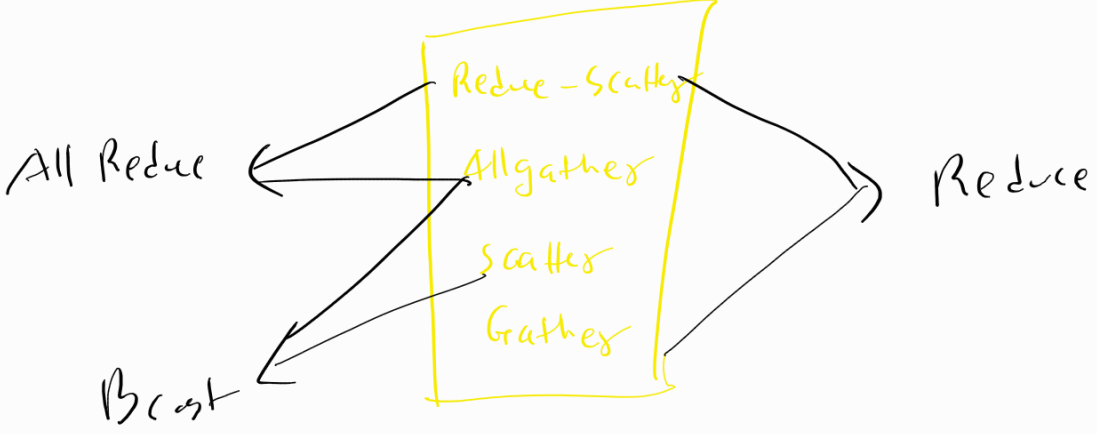
Note - To implementing these calls

↳ usually we use few building blocks using combination of which we can implement other calls

i) Spanning tree Algo



ii) Bucket Algo



Interconnects

metrices to be considered for a suitable interconnect

i) # links
ii) Diameter
iii) degree (max) } ↓ — better
iv) Bisection width → B.W ↑ — ↑ connectivity

Bisection width → min no of nodes to remove, that cause the network to be separated in two equal halfs → Decides cost of network → Also decides the lower bound

min volume of communication allowed b/w two halfs of a network
min no of links required to remove to create two halfs

	Linear	1D Torus	2D mesh	2D torus	d D torus	Hypercube	Trees	fat tree	Completely connected
# links	$p-1$	p	$2p$	$2p$	$2d$	$p\log p/2$	$2p$	$p\log p$	$p^2/2$
Degree	2	2	4	4	$p^{1/d}$	$\log p$	2	p	p
Diameter	$p-1$	$\frac{p-1}{2}$	$2\sqrt{p}$	$\sqrt{p}$	$\frac{p^{1/d}}{2} \times d$	$\log p$	$2\log p$	$2\log p$	1
Bisection width	1	2	$\sqrt{p}$	$2\sqrt{p}$	$2 \cdot p^{\frac{d-1}{d}}$	$p/2$	1	$p/2$	$p^2/4$

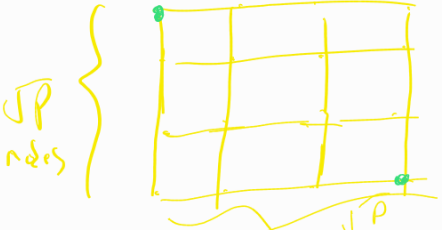
i) Linear

p nodes

ii) 1-D torus

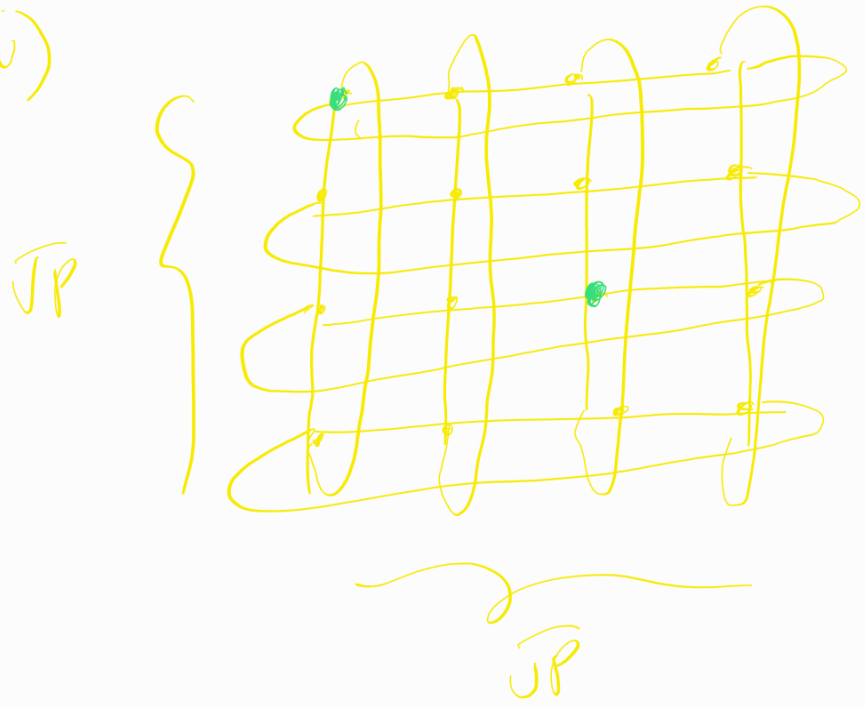
p nodes

iii) 2-D mesh



p processors

iv)



v) General formula for d-dimensional Torus & P-nodes

degree = 2d

links = $\frac{2d \times P}{2} = Pd$

diameter = $\frac{P^{1/d}}{2} \times d$

Bisection width = $2(P^{1/d})^{d-1}$

vi) Hypercube

↳ make nodes in a dimension = 2

$2 \times 2 \times 2 \times \cdots \cdots 2$ (d times) → P nodes

$\therefore 2^d = P$

$\therefore d = \log P \Rightarrow$ Hypercube

0-D Hypercube : •

1-D Hypercube : 0 — 1

2-D Hypercube :

3-D Hypercube :

degree : $\log P$

links = $P \log P / 2$

diameter : $\log P$

bisection width = $P/2$

To know if any random node is connected to which node

e.g. 100

→ check all P^n by changing 1 bit

000, 110, 101

e.g. 111 → complement 000

to check farthest node from a bit is 1

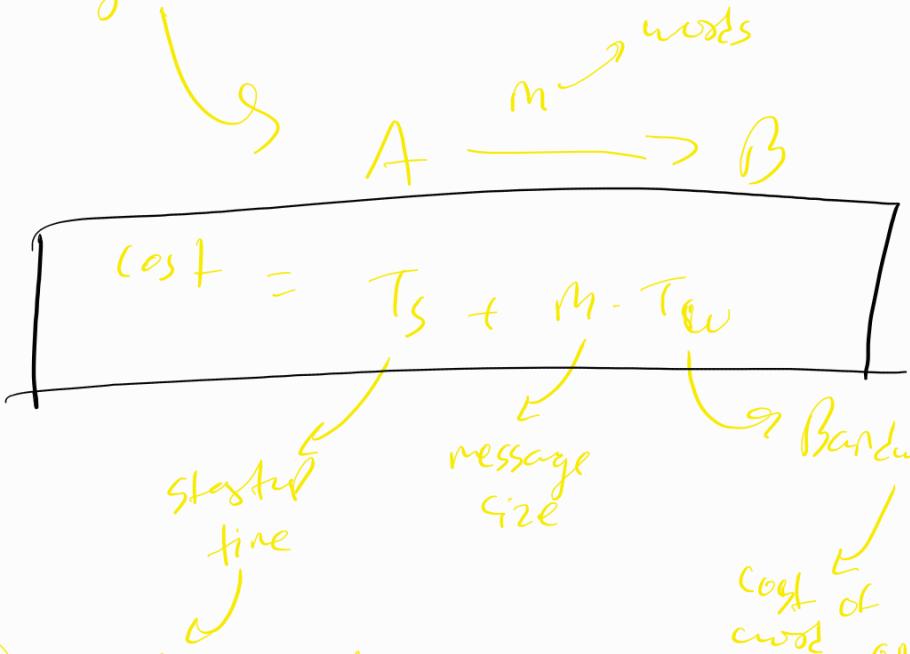
vii) Trees

viii) Fat Tree

↳ as we go up we keep doubling the links

* Evaluation of communications in ∥ Programs

1. Hockney models



before sending actual data A actually gives a trailer of data to B of what's coming & once B also shows it's ready to receive hence process begins of transferring

→ Code latency and transmission and transmission of header (kT_w)

- U can do only 1 send & 1 recv simultaneously
- communication with any node has same T_s
- Duplex: all links are bidirectional

now we'll try to analyse cost of few collectives

i) Bcast : 0 ... p-1

&

Reduce

p nodes

if 0 → 1 → 2 → 3 ... p-1

cost = $(p-1)(T_s + mT_w)$

very expensive

Here, recursive Doubling / Halving

0 → p/2

$0 \to \frac{p}{4}$, $\frac{p}{2} \to \frac{3p}{4}$

p = no of nodes

∴ log p = no of bits

e.g - if p=16 ⇒ no of bits = 4

0000

∴ $y_1\ y_2\ y_3 \ldots y_{\log p}$

0000 → 1000

here we going in opp. dirⁿ as it's convinient to only flip one bit

2nd bit from left flipped

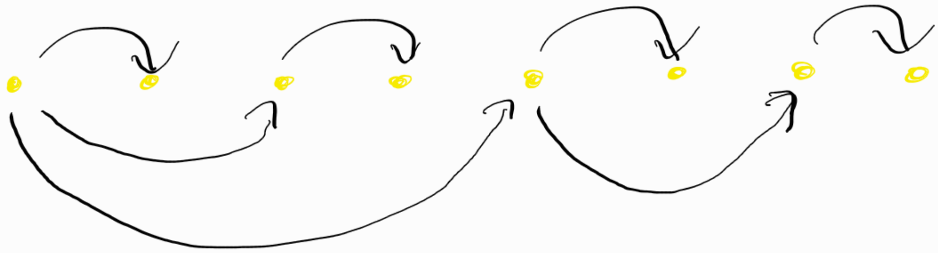
0000 → 0100 1000 → 1100

0

cost = $\log p\,(T_s + m \cdot T_w)$

by Recursive doubling

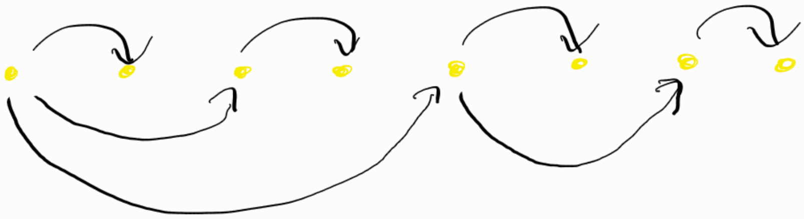
no of steps



$\therefore \quad cost = \log p (T_s + m \cdot T_w)$

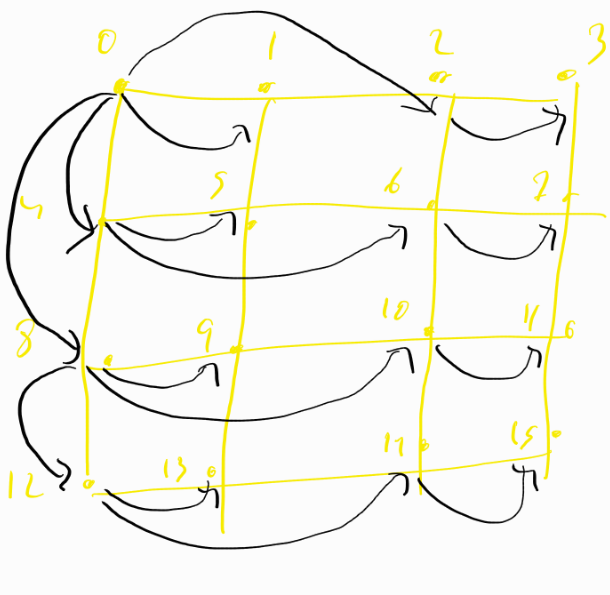
let's check on different architecture

a) linear

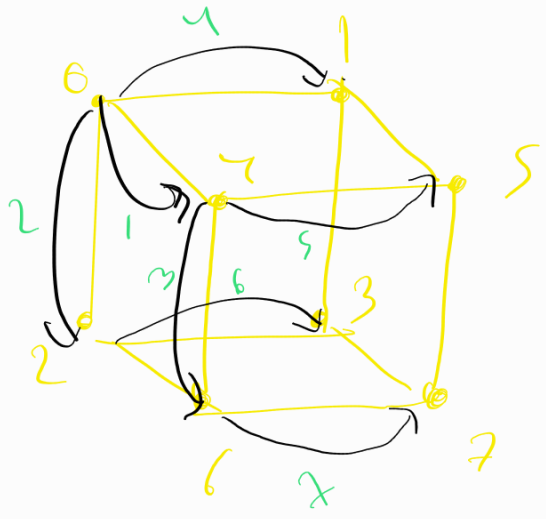


$\therefore \quad cost = \log p (T_s + m \cdot T_w)$

b) 2D mesh ($\sqrt{p} \times \sqrt{p}$)



c) Hyper cube



d) tree

Conclusion – cost remains same for all architecture.

Note – The cost will be same of Reduce as Bcast as it's just reverse of Bcast, Hence deal of Bcast

$$\therefore \text{cost of Reduce} = \log p\left(T_s + m \cdot T_w\right)$$

ii) Scatter & gather

send buff: mp — p nodes

recv buff: m m m m ... — p nodes

$$\therefore \text{cost} = T_s + \left(\frac{mp}{2}\right) \cdot T_w + T_s + \left(\frac{mp}{4}\right) \cdot T_w + \cdots$$

$$\text{cost} = T_s \log p + m\left(\frac{p}{2} + \frac{p}{4} \cdots 1\right) T_w$$

$$\boxed{\text{cost} = T_s \log p + (p-1) m T_w}$$

→ same for all architecture

iii) Reduce Scatter

&

All gathers : cost(Reduce) + cost(Scatter)

$\log p(T_s + mT_w) + T_s \log p + (p-1)mT_w$

$\therefore$ cost $= 2T_s \log p + 2(p-1)mT_w$

Can be optimized further

by recursive doubling

$\Downarrow$

cost $= (p-1)T_s + m(p-1)T_w$

Note— which approach is better to calc cost depends on T_s & T_w aka the size of the msg, if msg size big

$\Downarrow$

recursive doubling approach

if msg size is small, latency term dominates,

$\Downarrow$

Reduce + scatter

Note — Different sizes of words

Case1: size of the words remain same, then (m)

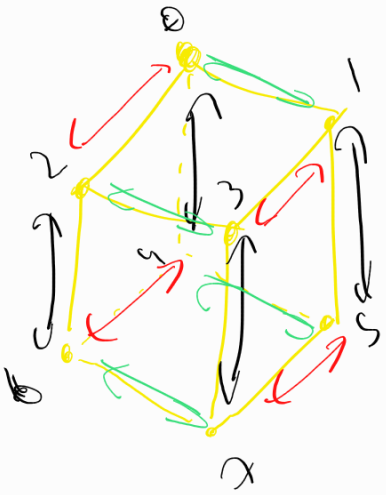
Case2: sizes of some/all nodes are diff, then (mp)

- Broadcast: m
- All reduce: m
- Scatter: mp → m => mp
- Reduce Scatter: mp → m → mp
- Gather: m → mp => mp

On mesh architecture

$$\text{Cost} = 2(\sqrt{p}-1)T_s + m(p-1)T_w$$

On Hypercube architecture:



element
original node element

0: 0[0] 1[0] 2[0] ... 7[0]

1: 0[1] 1[1] 2[1] ... 7[1]

2: 0[2] 1[2] 2[2] ... 7[2]

3:

4:

5:

6:

7: 0[7] 1[7] 2[7] ... 7[7]

iteration 1 : 0: 0[0+4], 1[0+4], 2[0+4], 3[0+4]

4: 4[0+4], 5[0+4], 6[0+4], 7[0+4]

1: 0[1+5], 1[1+5], 2[1+5], 3[1+5]

5: 4[1+5], 5[1+5], 6[1+5], 7[1+5]

2: 0[2+6], 1[2+6], 2[2+6], 3[2+6]

6: 4[2+6], 5[2+6], 6[2+6], 7[2+6]

3: 0[3+7], 1[3+7], 2[3+7], 3[3+7]

7: 4[3+7], 5[3+7], 6[3+7], 7[3+7]

iteration 2 : 0: 0[0+4+2+6], 1[0+4+2+6]

2: 2[0+4+2+6], 6[0+4+2+6]

1: 0[1+5+3+7], 1[1+5+3+7]

3: 2[1+5+3+7], 3[1+5+3+7]

5: 4[1+5+3+7], 5[1+5+3+7]

7: 6[1+5+3+7], 7[1+5+3+7]

4: 4[0+4+2+6], 5[0+4+2+6]

6: 6[0+4+2+6], 7[0+4+2+6]

logP iterations

iteration 3: 0: 0[1+2+3+4+5+6+7]

1: 1[1+2+3+4+5+6+7]

⋮

7: 7[1+2+3 . . . 7]

∴ Cost on hypercube for Reduce-scatter $= t_s \cdot \log p + t_w \cdot m \cdot (p-1)$

iv) All Reduce :

using hypercube architecture -

Note- same iterations just the message size changes.

∴ cost on hypercube for AllReduce $= (t_s + m t_w) \log p$

v) All to All → kind of like Transpose of matrix

By again Analysing Iteration of Hypercube with some iterations we can analyse cost just the size of message changes

$$\text{Cost on Hypercube for All to All} = \left(t_s + \frac{m}{2} t_w\right) \log p$$

Note- Conclusion :

Broadcast : $(t_s + m t_w) \log p$ [R.D] → to optimize we can pipeline → P parts → $(2p-2)t_s + 2m\left(1-\frac{1}{p}\right)t_w$

lower bound = mt_w

Scatter : $t_s \log p + m(p-1) t_w$

lower bound = mt_w

Reduce – scatter : $2 t_s \log p + m t_w (p \log p + p - 1)$ [R.D]

Ring : $(t_s + t_w \cdot m)(p-1)$

mesh : $2 t_s (\sqrt{p} - 1) + t_w \cdot m (p-1)$

Hypercube : $t_s \log p + t_w m (p-1)$

lower bound = $mt_w(p-1)$

All-Reduce : $2(t_s + m t_w) \cdot \log p$ [R.D]

Hypercube : $(t_s + m t_w) \log p$

lower bound mt_w

optimal → Chain : $2 T_s (p-1) + 2 t_w \cdot m \left(1 - \frac{1}{p}\right)$

All to All : Hypercube : $(t_s + m t_w) \log p$

E-cube routing $(p-1)\left(t_s + \frac{m}{p} t_w\right)$

lower bound = mt_w

Note- Again to compare which approach is better it depends on size of message

Short message → Recursive Doubling

Long message → Other Algo

Note- The lower bound for All to All was $\Omega(tw)$ but from Hypercube architecture we're getting a lower bound of $\Omega(tw)$ bgp which isn't correct so a new approach needs to be found

↓

E-cube Routing

↳ simplest, deadlock-free way to send a Package from any source S to any destination D in an n-dimensional hypercube. Works by fixing one different bit at a time

for AllToAll $(p-1)\left(t_s + \frac{m}{p} t_w\right)$

n cube ⇒ 2^n nodes ⇒ n bit binary string

e.g- if 3-D Hypercube, iteration = 5, map the nodes → $2^3 = 8$ no's

5 ⇒ 101

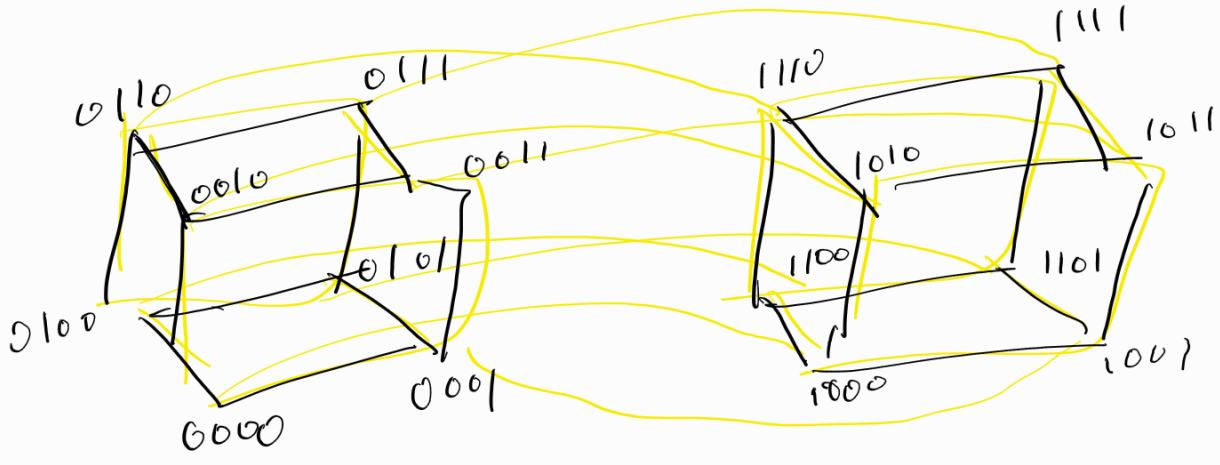
e.g. Find the Route Q.

4-D Hypercube

or

4-bit binary

S = 0110
D = 1101



→ Step-1 ⇒ $S \oplus D$ ⇒ $0110 \oplus 1101$

$= 1011$

$0110 \rightarrow 0111 \rightarrow 0101 \rightarrow 1101$

∴ Path

↳ 3rd bit from LSB will flip