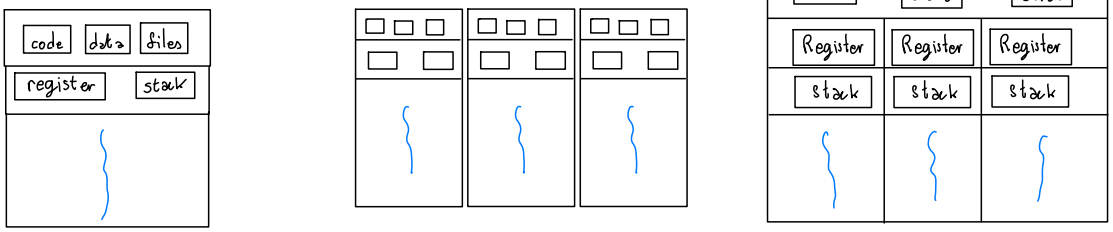
CLASS 3

Open MP : for parallelism in a multiprocessor computer by creating multiple threads

History of OpenMP

develop & accepted in 1990s as a standard which consist of compiler with library routine & environment var.

! use thread-based shared memory model



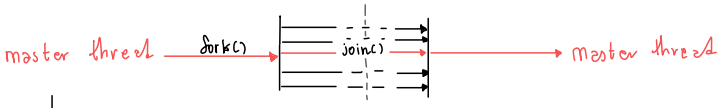
Single thread process | Multi Processing | multithread Processing

Fork / Join

Since OpenMP use multithread processing approach → the process can fork / join threads

Fork : master thread create parallel child thread *run code after fork() line

Join : complete the thread process return to parent thread



Writing OpenMP

```
# include < omp.h >
int main () {
    int var1, var2, var3;
    <sequential code>
    #pragma omp <directive> private (var1, var2) shared (var3) num_threads (_)
    {
        <parallel code>
    }
```

compile : gcc -fopenmp -o < > < .c >

what function to use ex. parallel, for

if not declared = global, if declare in parallel region = local

! '{' must in new line

thread will join here & auto add barrier for synchronization

```
int tid = omp_get_thread_num();          — return ID of each thread
int nthreads = omp_get_num_threads();    — return no. of threads created
```

Work can be shared Manually or use Directive Functions:

```
# pragma omp parallel private (i) num_threads (T)
{
    int part size = N/T
    int tid = omp_get_thread_num();
    int START = tid * part size;
    int END = (tid +1) * part size -1;
    -----
}
```

```
# Pragma omp for ...
for (i=0; ... )
```

"every thread do some part" in the for loop by splitting the work

! simple for loop (++, --, +=, -=)

Note if `pragma omp parallel` `pragma omp for` can combine to `#pragma omp parallel for`

```
#pragma omp Sections
{
    #pragma omp section
    < code >
    #pragma omp section
    < code >
}
```

execute separately by free thread

! some times it runs in Sequential

give compiler a hint that sections can run in parallel

```
# Pragma omp parallel Sections
{
    # pragma omp section
    < code >
}
```

ensures parallelism

Use Schedule to control work distribution

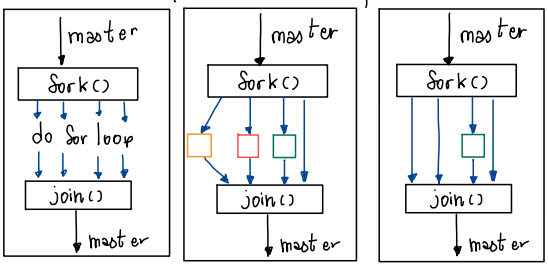
ex. ... for schedule (<type>)

..... for schedule (<type> , <chunk size>)

- Static : split equally using round robin
- dynamic : adjust size based on thread speed (default = 1)
- guided : initialize big chunk then decrease exponentially

```
# pragma omp single   → execute only 1
# pragma omp master   → execute only master thread (0)
```

omp for | omp sections | omp single



Private Variable

even we declare before #pragma...

Private: create private non initialized copy for each thread

Firstprivate: create private initialized copy for each thread
declare before the #pragma...

last private: set value = last value from the parallel region

Reduction Clause

use to combine local copies of variable in different threads

use with (parallel, for, sections)

operators: +, *, -, &, |, ^, &&, ||, max, min

ex. sum = 0

```
#pragma omp parallel for reduction (+: sum)
for (k=0; k<100, k++) { sum = sum + func(k); }
```

Race condition

#1 write after write

a ← R_1	b ← R_1
a ← a + 1	b ← b + a
R_1 ← a	R_1 ← b

#2 write after read

a ← R_1	
b ← a + 1	
	c ← R_1
R_1 ← b	
	d ← c + 1
	R_1 ← d

solve by Synchronization which is specifying a critical region to avoid race condition or using barrier to wait for slower thread

Omp Critical: only allow 1 thread to execute

```
#pragma omp critical [name]
<code here>
```

[name]: not allow same name execute together (optional)

Note: beware of using too many synchronization ≈ serialize

Omp barrier: wait until all thread reach the barrier

```
#pragma omp barrier
```

CLASS 4

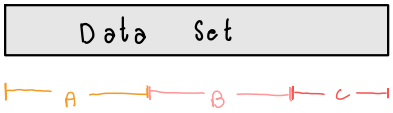
MPI Programming

what to consider when designing Parallel program

- degree of parallelism : how many simultaneous process
- initialization : what is required to run
- work distribution : who does what
- Data / IO access : where to access / work part
- communication : sharing information
- Synchronization : what to do after it is done
- Finalization : data collation between each thread

Work distribution

Type 1 : Data decomposition



- divide into pieces for repeated operation → ex. for loop

! no overlapping → consider dependency

Type 2 : functional decomposition

- divide the computational steps ex.

calculate A → MUL B → Add C

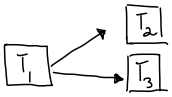
Dependency Analysis : find what can execute together

↳ Bernstein's Condition

1. input of P_1 not in output of P_2 → $I_{P_1} \cap O_{P_2} = \phi$
2. input of P_2 not in output of P_1 → $I_{P_2} \cap O_{P_1} = \phi$
3. no overlapping in output of P_1 & P_2 → $O_{P_1} \cap O_{P_2} = \phi$

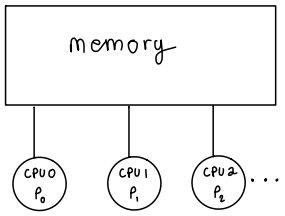
↳ task graphs : use to show the dependency

; node = task, edge = dependency



Parallel Programming Model

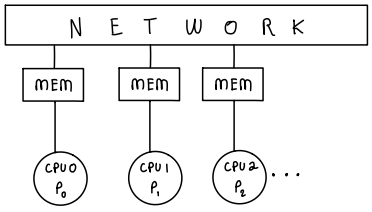
#1 Shared Memory



All task access the same memory space

(threads, openMP, fork())

#2 Message Passing



uses local memory → to share data must transport it explicitly

(MPI, MPICH)

MPI

Message Passing Interface

- free implementation
- support multiple language
- support multiple Architures (parallel pc, distributed system)

Provide 2 things

- method to create seperate process
- method of sending & receiving msg.

Creating MPI Programs

Compiling → $ mpicc -o <filename> <filename>.c

Running → $ mpirun -np <no.of process> <executable name> <arg ..... >

Function	Description
int **MPI_Init**(int *argc, char **argv)	Initialize MPI
int **MPI_Finalize**()	Exit MPI
int **MPI_Comm_size**(MPI_Comm comm, int *size)	Determine number of processes within a comm
int **MPI_Comm_rank**(MPI_Comm comm, int *rank)	Determine process rank within a comm
int **MPI_Send**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Send a message
int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)	Receive a message

* the process execute concurrently
* msg passing send data between process
* msg have tag to tell the type

know who are you [0, process - 1]

Communicator

NOTE : Id start at 0

communicator : MPI_COMM_WORLD declare the entity where communication between process can occur

initialization

```
main () {
    int  rank, size;
    MPI_Init (& argc , & argv)   // no need to be 1st statement
    MPI_comm_rank ( MPI_ comm_WORLD , & rank );
    MPI_Comm_size ( MPI_ comm_WORLD, & size);
    <code>
    MPI_Finalize ();
}
```

! the variable is independent

↑ "SMPD" single program multiple data

Sending & Receiving data

. what to send	. what to receive
. where to send	. where (from whom)
. How many to send	. How many

MPI Send

MPI_Send (<address of what to send> , <amount> , <data type> , <destination rank id> , <tag> , <MPI_comm> , <MPI_Status status>);

(* pointer; int; MPI_DATATYPE; int; int; MPI_Status)

→ All process will beblocked untill data transfer is done

MPI_Receive

MPI_Recv (<address where to store> , <amount> , <data type> , <source rank ID> , <tag> , <MPI_comm> , <MPI_status>);

* status can be use in many way : error code, msg. tag, sender rank, etc.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

1D Array distribution = send & A , length

2D Array distribution = send &A[0][0], length

(-) 1 free master process that wait for response

CLASS 5

Collective Communication

- 1 function but call by every process
  - * no msg. tag
  - * sometime better than send/recv

MPI_Bcast() : broad cast value from root to all other process — Recv is not necessary !

MPI_Gather() : gather value from set of process

MPI_Scatter() : scatter buffer into small part and give it to group of process

MPI_All to all() : exchange data of all process (every one have all value)

MPI_Allgather() : gather and give it to all process

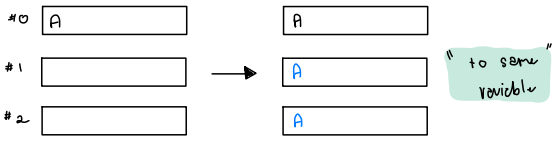
MPI_Reduce() : combine value from multiple process to one

MPI_Reduce_Scatter() : combine value and give it to all process

MPI_Barrier() : wait until every process reach this checkpoint

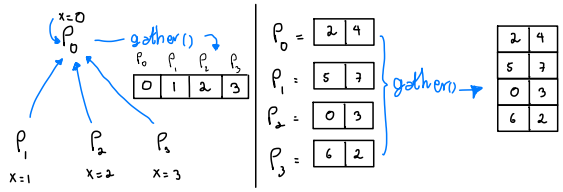
Bcast

```
Bcast ( *data, size, type, source id, MPI_comm );
```



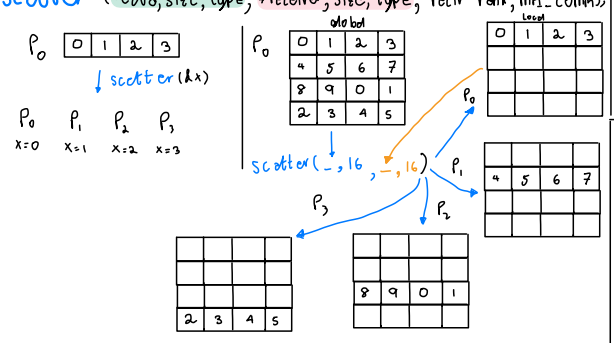
Gather

```
Gather (*data, int size, type, *receiver, int size, type, int reciv rank
MPI_comm);
```



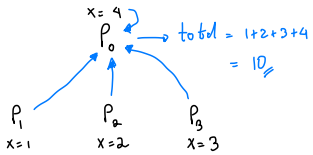
Scatter

```
Scatter (*data, size, type, *receiver, size, type, recv rank, MPI_comm);
```



Reduce

```
Reduce (*data, *result, size, datatype, reduce function,
destination process id, MPI_comm);
```



ex. implement in integral function

Array Reduction

MPI_Reduce (Min) → 0 2 ! compare 1 address of each process
root = 0

MPI_Reduce (Sum) → 13 16
root = 1

Allgather

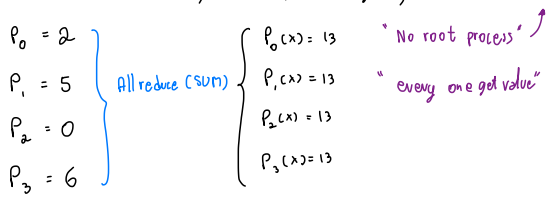
```
Allgather (*data, size, type, *receive, size, type, MPI_com);
```

"every one get data"

sorted by rank

Allreduce

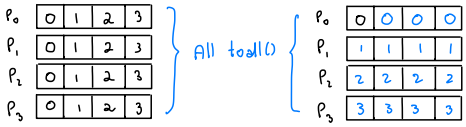
```
Allreduce (*data, *result, size, type, OPERATOR, MPI_com);
```



" No root process "

" every one get value"

All to All



" work in column"

Barrier

```
Barrier (MPI_COMM);
```

wait until every process complete

CLASS 6

Performance Evaluation

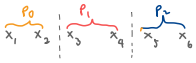
goals

- Portion the work equally → "load balance" (reduce synchronization time: wait time for another process)
- Reduce Overhead ~ penalty of setting up before execution ex. sending data

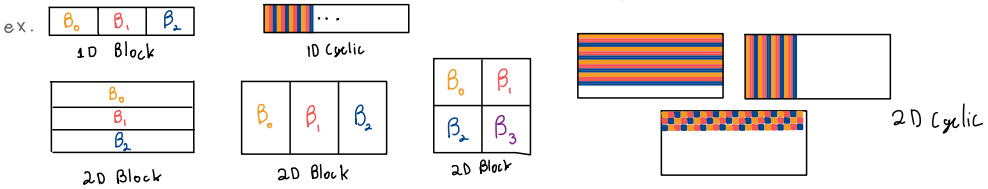
Data decomposition

Interleaved (cyclic): use mod function to provide each process data →

Block: give process a continuous set of data



ex.



What to do if N ∤ Process

for interleaved → for (i=pid ; i<n ; i+=p)

for Block → calculate range & owner of that range

START → $\lfloor \frac{pid \times N}{p} \rfloor$

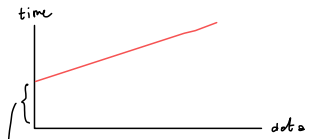
END → $\lfloor \frac{(pid+1) \times N}{p} \rfloor - 1$

} OWNER → $\lfloor \frac{p(elemID+1)-1}{N} \rfloor$

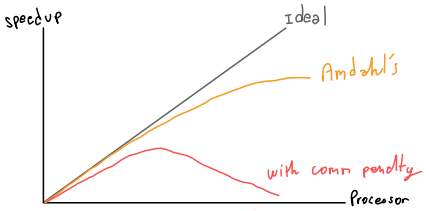
Performance Concepts

Execution time

- Sequential (t_s): calculate by counting steps
- Parallel (t_p): equal to slowest process
  - $t_p = t_{compute} + t_{communication}$ → $t_{startup} + nt_{data}$ (time for connection / startup + transmission time)



Speed Up ($S(p)$) = $\frac{T_{sequential}}{T_{parallel}} = \frac{t_s}{t_{comp} + t_{comm}}$



Ideally speedup should increase linearly (↑Process = ↑speedup)

But program sometime have sequential part → δ-factor "Amdahl's Law"

$S_{(p)} = \frac{t_s}{(\delta t_s) + \frac{(1-\delta)t_s}{p}} = \frac{1}{\delta + \frac{(1-\delta)}{p}}$; Upper limit = $1/\delta$

! overestimate: there is no parallel overhead penalty

"Amdahl's Effect" → large data set will eliminate time penalty ≈ linear

Gustafson's Law: calculate from ideal speedup then deduct serial penalty of idle process when run sequential code

Scaled Speedup" → $S_s(P) = \frac{st_p + P(1-s)t_p}{st_p + (1-s)t_p} = P + (1-P)s$ (Assume constant; s: time spent in serial section)

Amdahl: sequential → parallel estimation "constant problem size scaling"
Gustafson: parallel → sequential estimation "time constrained scaling"

Computation / Communication ratio: show the amount of communication required = $\frac{t_{comp}}{t_{comm}}$

Efficiency: show how effective of parallelism = $\frac{S(p)}{p} = \frac{t_s}{p \cdot t_p}$ } show improvement per processor

Timer in MPI

```
double time = MPI_Wtime()   // in seconds
```

CLASS 7

Not in exam!

CUDA Programming

Parallel programming using Processor Array Architecture

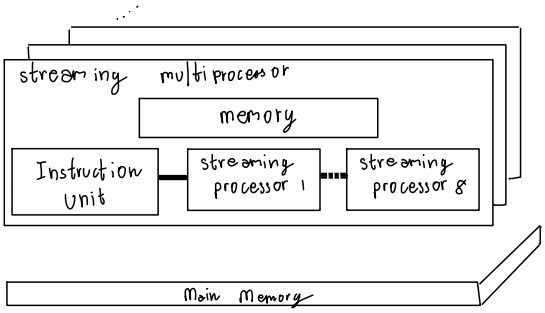
GPU: graphic Processing Unit → a processor array for graphic rendering

* can calculate faster than CPU
* have higher memory bandwidth (x10x)
* mainly consist of ALU support for simple & specific instructions
* communicate via PCI-e bus (Peripheral Component Interconnect - express)

GPU Architecture

#1 high throughput global memory → "device memory"

#2 set of Streaming Multiprocessor



Streaming Multiprocessor contain : work the same instruction [SIMD]

- instruction Unit
- group of streaming processor
- shared local memory

GPGPU

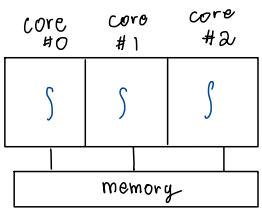
General Purpose graphic processing unit

- general computation using GPU and graphic API in application other than 3D graphic
- using in Parallel computation (large data array, SIMD parallelism, low latency of floating point calculation)

CUDA: compute unified device architecture

- a general purpose programming model by NVIDIA
- extend ANSI C programming language
- low learning curve

Shared Memory Programming



- Process have multi thread, flow of program, which can be construct by fork() but here we focus "CUDA threads" which will run in parallel

requires

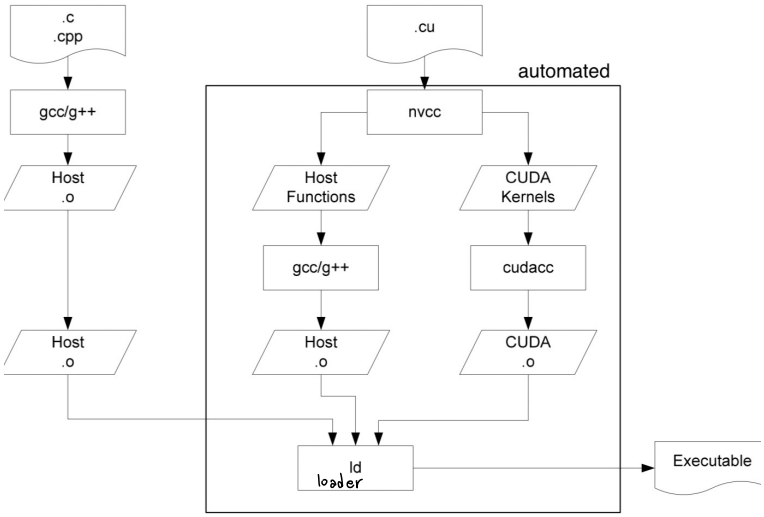
1. CUDA Driver & SDK
2. editor with CUDA source file (.cu)

Compile

```
>nvcc -o <out> <code.cu>
```

Run

```
>./out
```

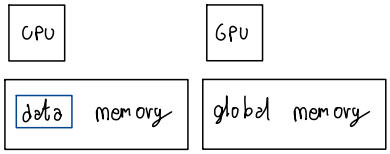


Writing Code

Initial

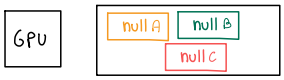
- data loaded to computer memory

```
int a[N], b[N], c[N];
//Init data of a,b,c
```



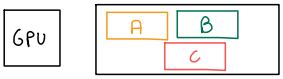
STEP # 1 Allocate GPU Memory

```
int *gpu A, *gpu B, *gpu C ; // pointer to GPU
                               mem location
cudaMalloc ((void**) & gpu A, size),
cudaMalloc ((void**) & gpu B, size),
cudaMalloc ((void**) & gpu C, size),
```



STEP # 2 copy data to GPU memory

```
cuda Memcpy (gpuA, a, size, cuda Memcpy Host To Device);
cuda Memcpy (gpuB, a, size, cuda Memcpy Host To Device);
cuda Memcpy (gpuC, a, size, cuda Memcpy Host To Device);
```



STEP #3 execute using kernel code

with CUDA syntax (no. of thread & physical structure)

```
myKernel <<< n, m >>> (arg 1, ... );
```

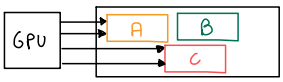
function — grid size: n block, m thread

```
N = 256;
```

kernel routine CUDA specifier

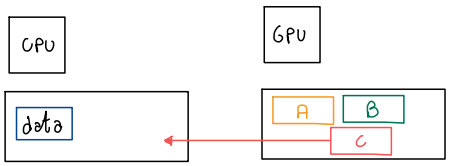
```
_global_ void vecAdd (int *A, int *B, int *C) {
    int i = threadIdx.x ;
    C[i] = A[i] + B[i];
}
vecAdd <<< 1, N >>> (gpuA, gpuB, gpuC),
```

256 threads to add
1 index simultaneously



STEP #4 transfer data back to CPU

```
cuda Memcpy (c, gpuC, size, cuda Memcpy Device To Host);
```



STEP #5 free GPU memory

```
cuda Free (gpuA);
cuda Free (gpuB);
cuda Free (gpuC);
```

if no. of thread < data

```
for (i = threadIdx.x ; i<n ; i+= T ) {
        C[i] = A[i] + B[i];
}
```

CLASS 7
EXTRA 1

Finding Prime Number → Sieve of Eratosthenes

- 1st prime is 2
- delete all number of stored prime tuples
- next number will be prime & repeat

sequential

List = 2, 3, ..., n

k = 2

while $k^2 < n$

- all no. of k tuples between $k^2 - n$ is marked
- k = smallest un marked

un marked = prime

Make it Parallel

♡ marking the non prime from known prime

↓

data decomposition by break to sub array

$first = \lfloor \frac{pid(n)}{p} \rfloor$, $last = \lfloor \frac{(pid+1)n}{p} \rfloor - 1$

$owner = \lfloor \frac{(p(i+1) - 1)}{n} \rfloor$

Note

- we sieve to $\sqrt{n}$
- ∴ P should smaller than $\sqrt{n}$ so it contain all prime

👍 no need to reduce data to P_0

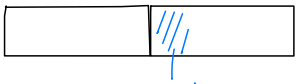
Parallel the marking

P_0 broadcast prime to other process to mark

```
for (i = first ; i ≤ last ; i++) {
    if  i % K == 0  → mark i
```

! % slow
* is faster

```
if first % k = 0 → mark(first, first + n k)
    untill it reach last value of it's block
```



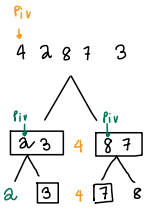
Local = 0
global = 3

Find Pseudo code

1. create list [2, n]
2. k = 2
3. repeat untill $k^2 > n$
   - mark all k between $k^2 - n$ ← multiple thread
   - k = smallest un marked → P_0 only
   - P_0 broadcast K
4. un marked is prime
5. reduce the result to prime only

CLASS 7
Extra 2

Parallel Sorting



Optimal speed $O(n \log n)$ of sequential
∴ ideal for parallel = $O(\log n)$

Sequential Quick Sort

1. choose Pivot value
2. split the data into 2 group
   Lower than pivot, Higher than pivot
   ! pivot value is in the right position

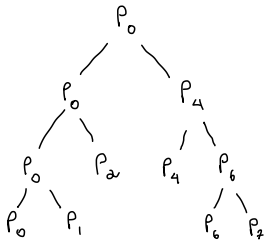
repeat

Average:	$O(n \log n)$
Worst:	$O(n^2)$

unbalanced at every step

Parallel quick sort

♡ use Process tree to handle the splitted data section
simple → use $P = n$ → impossible



but $n \log n$ is not guarantee because splitted array might unbalance
↳ choose better pivot "Median of 3" : pick 3 number from data and choose the middle one

Computation time (t_{comp})

$$t_{comp} = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \approx 2n$$ ← measure using computation step

Communication time (t_{comm})

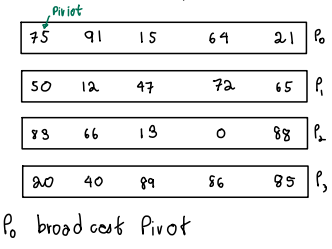
! no. of process $= 2^k$

1st $P_0 \rightarrow P_4 = t_{start} + (n/2)t_{data}$
2nd $P_0 \rightarrow P_2$, $P_4 \rightarrow P_6 = t_{start} + (n/4)t_{data}$
⋮

total $= (\log n)t_{start} + n t_{data}$

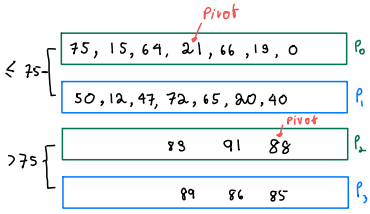
! to begin next step must wait all process to finish current step
! picked pivot might not be the "True Median"

ex. initial : data split to all process



P_0 broadcast Pivot

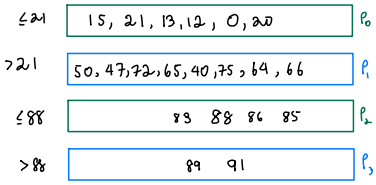
STEP 1 : pair P_0-P_2, P_1-P_3 & exchange low/high



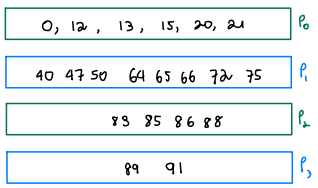
! execution time
= last process finish

! pivot effect balancing

STEP 2 : pick pivot of P_0, P_2 & exchange



step 3 : each processor sort it's own value



Merge Sort

Sequential : $O(n \log n)$

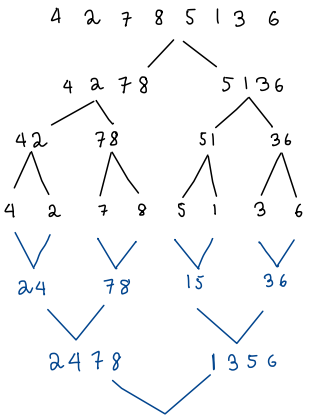
"Divide & Conquer"

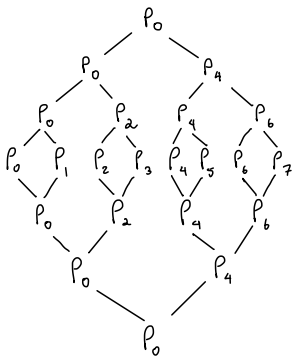
; split unsorted list into halves untill only 1

then merge into sorted by each branch

Parallel : $O(2 \log n)$ → log n for splitting

→ log n for merging





Merging Step

- use 2 index to point data in each list
- put the lower data in another list
- total = $m+N-1$ comparison

C[ ] ← A[ ] , B[ ]

COMPLEXITY → 2 log n steps

↳ $2 \log n (t_{startup}) + 2n t_{data}$

[communication]

$t_{comp} = \sum_{i=1}^{\log n} (2^i - 1) \rightarrow O(n)$; n processor

(compute only in merging phase)

[computation]

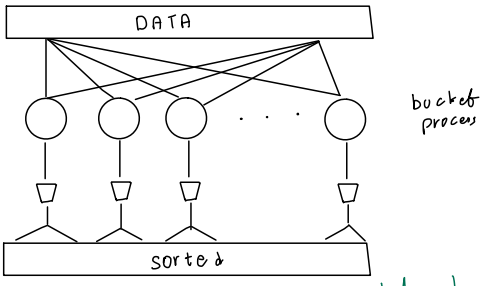
Bucket Sort

each "bucket" have an acceptance range

1st Broad cast data to every bucket process

2nd Pick only value in range of the bucket

3rd Sort locally



COMPLEXITY

broad cast → $t_{comm} = t_{start} + n t_{data}$

merge (1-1) → $(P-1)(t_{start} + n/p\, t_{data})$

compute = n + $(n/p) \log (n/p)$ (assume balanced)

bucket filter, sorting inside = (n log n)

Rank Sort

calculate rank from counting lower data in data set

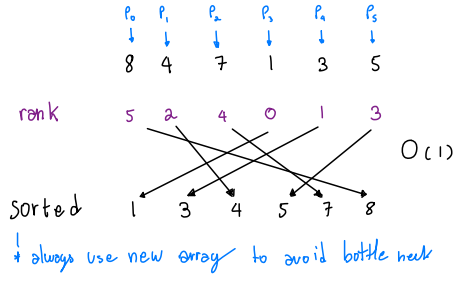
rank = position on the sorted array

1st broad cast data to every process

2nd each process pick value based on process id

3rd compare all data rank + 1 is there is data lower

4th insert it to result [rank]



! always use new array to avoid bottle neck

Sequential = $O(n^2)$; every one compare

Parallel = $O(n)$

COMPLEXITY

broad cast → $t_{comm} = t_{start} + n t_{data}$

merge (1-1) → $(P-1)(t_{start} + t_{data})$

compute = n - 1 comparison