High Performance Computing

Performance →

Algorithms →	Computer
Deterministic	Non deterministic
Abstract	Not abstract
Always Terminate	Might not terminate.

Unit of Performance - Flops / Sec

= Frequency x Number of Cores x Flops per cycle.

= $\frac{2.5 \times 10^9 \text{ cycle}}{\text{cycle sec}} \times 4 \text{ Flops / cycle}$

= 10 GFLOPS / ~~cycle~~ Sec ← Serial Programming

(output that is obtained theoretical Limit)

Performance is determined by - 1) Architecture → Cache
2) Algorithm
3) Software Environment

- Scalability - Able to utilise higher number of cores
- Portability - Must run on other architectures too.

For calculation of performance of a code, it must take into account the memory access time required (As we have a lot of data which needs to be loaded)

- Systems Perspective - Architecture, ~~Algorithm~~ Software Environment and ~~Software Environment~~ Algorithm
  (Non deterministic) (Deterministic)

Get

= Max possible performance out of a particular system for a particular problem

- A node can have one or more sockets
  Each socket has a processor
  Each processor can have multiple cores.
  Each core performs FLOPs

Several such connected nodes = Supercomputer

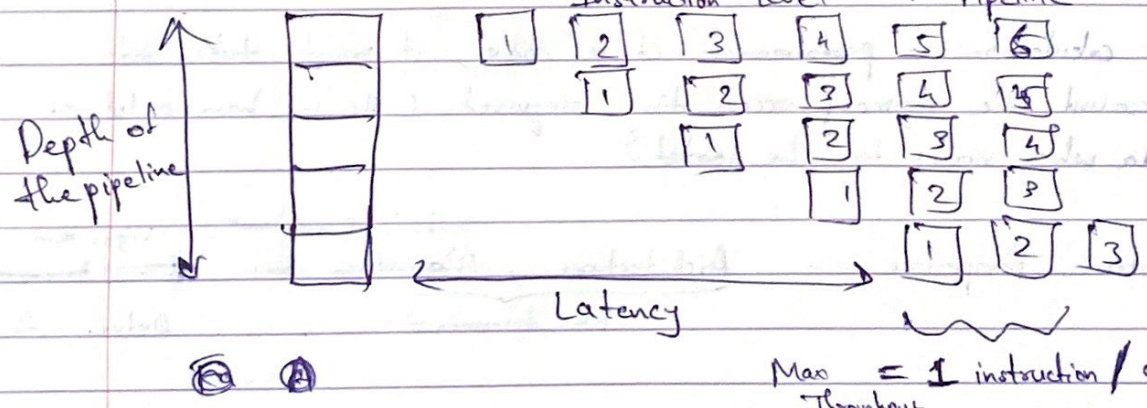
* Properties of Memory - Latency (second) and Bandwidth → (MB/s)
Increases in order Secondary, RAM, Cache, Register to Processor

- Serial - Only one instruction at any moment in time.
  Improve latency (How fast you can compute/1 instr
- Parallel - Solving concurrently
  Throughput is good (How fast n instructions can be compu

- Speedup = $\frac{\text{Serial Time}}{\text{Parallel Time}}$

* Multiplicity :- There are multiple memory
  - Multiple data paths
  - Multiple processor

- Types of parallelism - Node Level
  Thread Level
  Instruction level → Pipeline



- 1 result / cycle after wind up phase
- Depth of pipeline = m stages
- Total independent operations = N
- Total time of the pipe line = N + m - 1
  (Without pipeline) - Serial = mN

Speedup = $\frac{mN}{N+m-1}$

If $N \gg m$ then Speedup = m

Throughput of the pipeline = $\frac{N}{N+m-1} = \frac{1}{1+\frac{m-1}{N}}$

when $N \gg m$, Throughput $\rightsquigarrow 1$.

Thus for ~~ss~~ smaller problem size, throughput won't be high hence there would be lower performance levels.

- If P_1 requires 40 bytes of data; P_2 requires 1 KB, P_3 requires 1 MB then all of them will take the same access time.

  Latency = Time required to access 0 bytes ~~of~~ data.

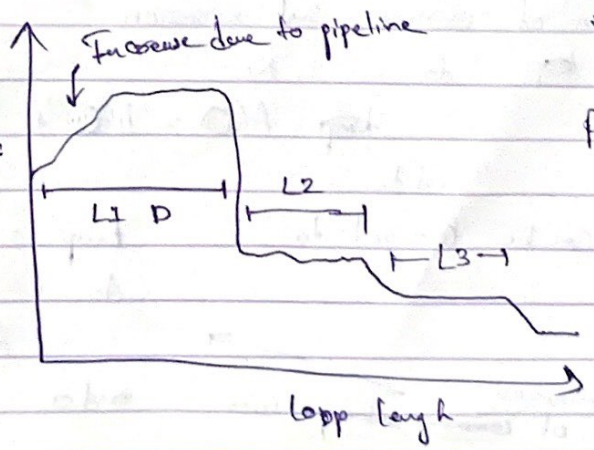
- Von Nuemann Bottleneck :- Not able to provide data to the process at the desired ~~ed~~ rate ie The rate at which processor process the instructions.

- Vector Triad :- 1 Multiplication + 1 addition - a + b * c

Q Plot the graph of Performance vs N

(Measure Time
Timer function for elapsed time
* Resolution of timers.

pipeline can not produce
of 1. thus it is low.
pline throughput to 1.
L1 is low
& L2 is higher
ck in bus channel



plot for single core

Addition and Multiplication have very short pipeline. Thus both

Machine Balance	Code Balance
- Machine Balance = Memory Bandwith / Peak performance Unit = GB / FLOPS - Differ~~ce~~ for cache, RAM, disk - Historically decreasing trend, will decrease in future too due to increase in peak performance (More cores)	Code Balance of loop = Data traffic / Number of operations = 4 (A B C D) / 2 = 2

- ~~Code Balance of Vector Triad~~ Reciprocal of code balance = Computational ~~Code~~ Intensity
- Lesser the value of code balance, more efficient will be the code.

Lecture Miss on 15th Jan

Granularity = Ratio of computation to communication
Coarse → Large computation betwe
Fine → Small computation betwe

Simple Optimization of Serial Code

- Using Exit → Lesser computations of non required things)
- A = A + B**2

Strength Reduction

A = A + B**2.0 ✗ → A + Exp(2 ln B)

A = A + B * B ✓ Better because pipeline of multiplication is lower. (Lower stages in pipeline)

First one is transcendental → very long pipe

- Declaration in tmp of things which is computed over and over

Elimination of common subexpression

```
do i = 1, N
   A(i) = A(i) + sin(x) * i
end do
```

can be changed to

```
tmp = sinx + A
do i = 1, N
   A = A + tmp * i
end do
```

Elimination of common sub expression

- Single precision vs Double precision
  Single is better as size of single is less, more things can be stored in cache ⇒ Higher Cache rate ratio.

Q. Which is better $B^k * B * B$ 10 times or B^{10}

- Avoid Conditional Branching → Reason - Branch Miss can happen.
  The compiler predicts that the result of if conditionals based on statistical algorithms of compiler. It will start calculation based on its assumption. But if the condition fails than it has to empty its pipeline and re start computation

```
do i = 1, N, 2
  do j = 1, N
    c(i) = a(i,j) * b(j)
    c(i+1) = a(i+1, j) * b(j)
  end do
end do
```

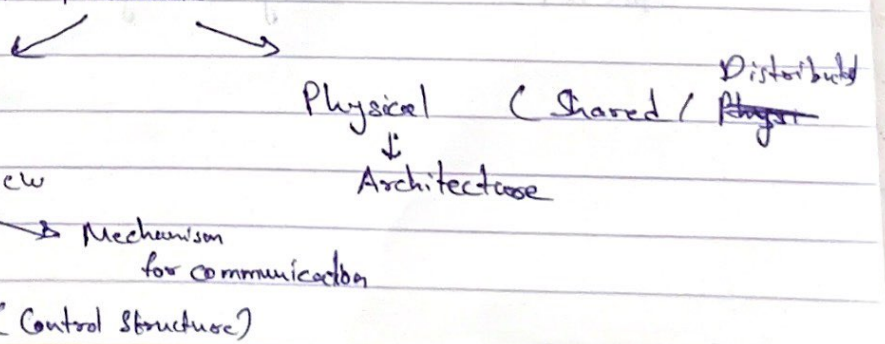
✓

```
do i = 1, N
  do j = 1, N
    c(i) = a(i,j) * b(j)
  end do
end do
```

✗

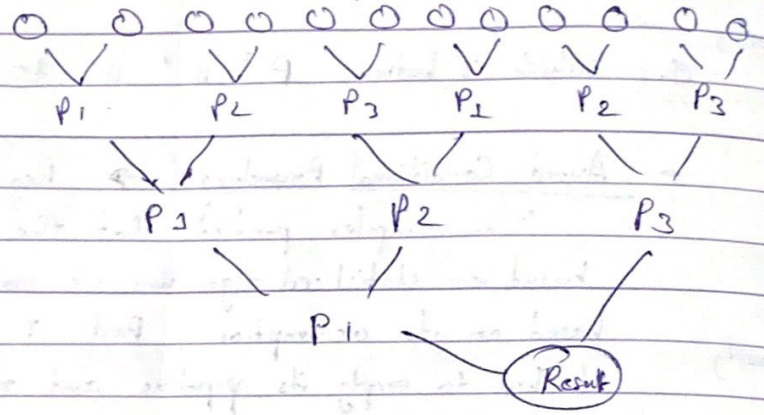
First is better because of Loop Unrolling. The c array will be loaded in the memory. ⇒ Higher cache hit.
If c(i) is loaded it will load c(i), c(i+1) ... (if N ≈ 10^6 or more)
if a(j,i) and b(j) loads up the cache then c(i) will be needed to be loaded again in the cache

- Organisation of Parallel Platforms

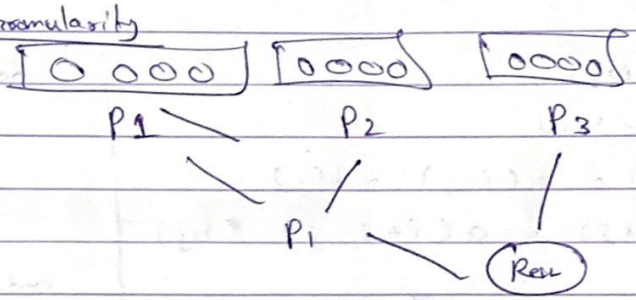


- Shared Memory Parallelism

- Fine Granularity for N addition (3 processors



- Coarse Granularity



For ~~multi~~ matrix multiplication of n x n matrix. There are 4 process
Finest granularity - n^2 (Each processor doing 1 multiplication of cell)

1 row 1 column by a processor → Granularity level - n
n/4 rows & n/4 columns by a processor → Granularity → n/4

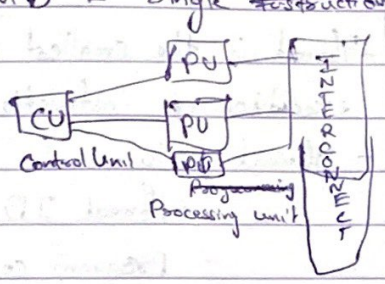
OpenMP → MP stands for multi processing
Open MPI → Message Passing → For Distributed System

Q 64 bit DDRAM interface, 8 channels, 1 GHz clock speed. Access latency of 200 cycles - ?

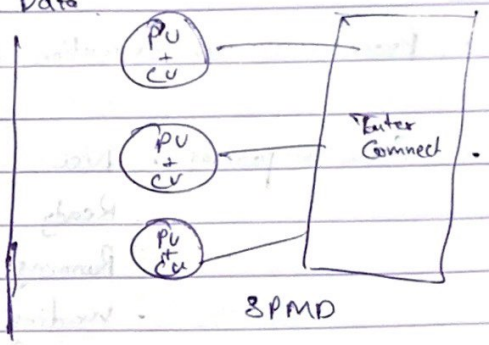
Ans Access Throughput (bytes/sec) = 8 bytes/transfer × 8 channels × 2 transfers/clock/chanel (Double Data.) × 10^9 clocks/sec

= 128 GBps

- SIMD → Single Instruction Multiple Data
SPMD - Single Program Multiple Data



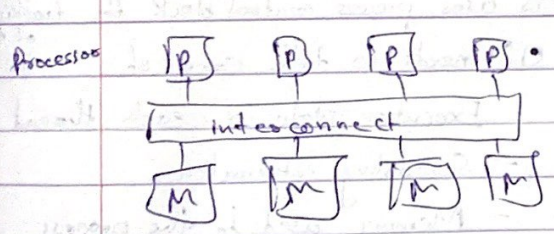
SIMD



SPMD

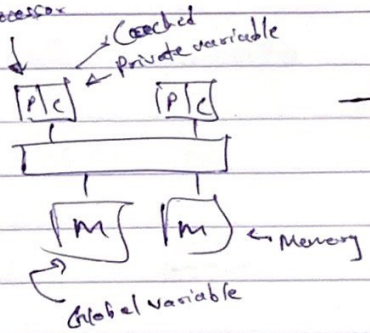
Q. Which is faster → Ans - Depends on the program

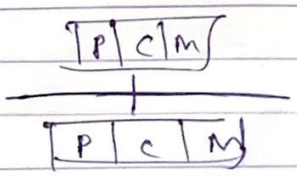
- Global Memory → Easy to write / Difficult to write.
- UMA → Uniform Memory Access | NUMA → Non-uniform Memory Acc



Each processor will take same time for process

(Shared)



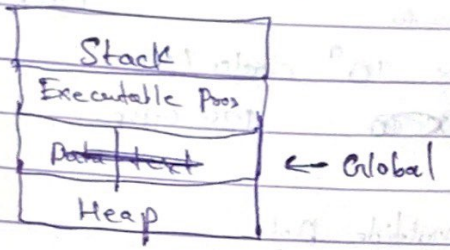


(Distributed)

Memory can be still accessed through interface but will require (at at time)

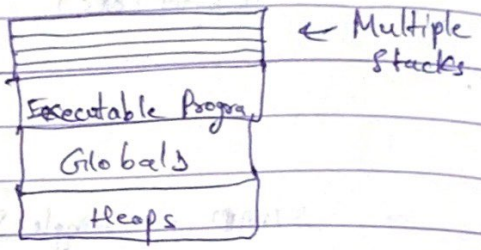
Stack is faster than heap

- **Process** :- Running Program
  There will be context switch
- Ath process has an id. - pid
- Status of process - waiting, running etc



- Process is an execution stream
- Status of process - New
  - Ready
  - Running
  - Waiting
  - Terminated

Threads :- Lightweight process
They run concurrently
Thread too has id



- Thread is the smallest unit of sequential execution of instructions
- Thread includes
  - Thread ID
  - Program counter
  - Register state (pointer)
  - Stack (pointer)
- Shared Resources
  - Text section
  - Data section
  - Other OS resources

OS uses process control block for tracking of threads
OS needs to take care of
- Execution state of each thread
- Scheduling information
- Memory used by the process
- Information about the open file

- Scheduling of SIMD is easier than SPMD because in SPMD there are multiple control unit which needs to be scheduled accordingly.

* Dependencies

```
for i = 1, n
    for j = 2, m
        b[i,j] = ... + b[i+j-1]      // j is dependent.
```

Data Dependencies

- Types of dependencies - Flow, Anti, Input & Output

Flow/True → ~~Stater~~

Statement 'i' precedes 'j' and computes the values that 'j' uses

① x = 1 | ② y = x + 2 | ③ x = 3 - w | ④ x = y / 2

Number of flow dependencies - ① → ②
② → ④

Anti - Statement 'i' precedes 'j' and ~~computes~~ uses ~~the~~ a values that 'j' computes

① x = 1 | ② y = x + 2 | ③ x = z - w | ④ x = y / z

② → ④
② → ③

Output - Statement 'i' precedes 'j' and 'i' computes a value that 'j' also computes

① → ③ | ① → ④ | ③ → ④

Input - Statement 'i' precedes 'j' and 'i' ~~computes~~ uses a value that 'j' also uses.

③ → ④

- If dependence flows from 'i' to 'j'

'i' → Source, 'j' → Sink

Flow / True dependency is difficult to remove

- In data dependency graph, nodes are the statements, whereas edges are the dependency relations

Eg - a[i] = b[i] + c[i]
d[i] = a[i]

There is flow dependency but no loop carried dependency
Dependence distance = 0

- a[i] = b[i] + c[i]
d[i] = a[i-1]

This has loop carried dependency → Dependence distance = 1.

```
for (i=0 ; i<N ; i++)
    a[i-1] = b[i]
for (j=0 ; j<N ; j++)
    c[j] = a[j]
```

Can be parallelized
Locality is worse

Loop distribution →
← Loop Fusion

```
for (i=0 ; i<N ; i++)
    a[i-1] = b[i]
    c[j] = a[i]
```

Cannot be parallelized
Locality is better

We use Loop Fusion so as to get better utilisation of spatial locality
We use Loop Distribution so that we can use parallelism.

- Directives :- Special preprocessor instructions

- For shared systems - All processes share common memory
  - Easy to code.
  - Bus contention leads to poor scalability

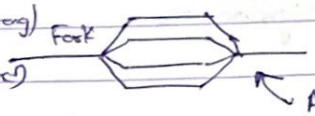
- In distributed, each processor has private memory
  - Scales well
  - Difficult memory management

- Fork - Join

  Fork is called by thread (parent) to create new thread (child).

  Fork helps in concurrency (increase concurrency)

  Join removes concurrency (decreases concurrency)



All must unite before continuing.

(~~Explicit~~ Barrier)
Implicit

  Parent must join with child

- A structure blocked :- There is one point of entry, one point of exit.

  Inside structure block if condition is not allowed.
  - In general branching is not allowed.
  - Check if exit() is allowed or not.

  A structured block can be executed in parallel. (mostly)

- Scope → Access (Who can access it)
  → how long is it visible.

- Performance of Shared Memory depends on
  - Load Balancing (mapping work loads with thread scheduling)
  - Caches
  - Locality
  - How locality affects scheduling algorithm

  ↳ 1003 to 4 threads
  250 each.
  The rest 3 can be given to the one who finish the 250 tasks first.

- Work Sharing directives just shares the work, does not create threads

```
# pragma omp parallel   ← thread create
# pragma omp for        ← work sharing.
```

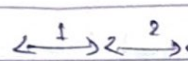
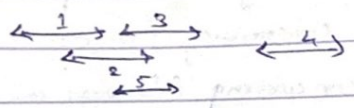
if the variable is in for loop then it is converted to private.

#pragma omp for
for (i=0 ; i<N ; i++)

- Since work sharing is used, the i will be taken as private to the thread

②

* Index variable of a worksharing loop is private by default.

- Synchronous -  (Block Mechanism)
- Asynchronous -  (Non-Blocking)

- Work sharing - for directive → shares iterations
  section directive →
  critical directive → serializes section of code.

→ For work sharing directives
  - No implicit barrier at entry. (No need to wait for other
  - Implicit barrier at exit. (Every thread waits for process to comp
  → No wait clause is used to remove this barrier.

→ Scheduling Mechanism for For → static, dynamic guided.
  Default size is 1. chunk.

- Thread Synchronisation :-
  Implicit Event Synchronization
  Explicit Synchronisation - critical, master directives in OpenMP
  - Critical
  - atomic, single, master, barrier, flush, omp_set_lock()

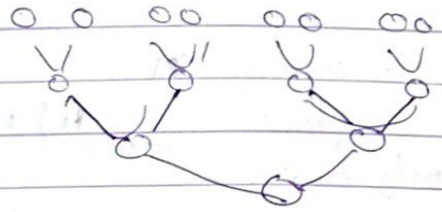
- OpenMP - Reduction (Floating point operation ≠ Associative)
  - It is binary operator.
  - ~~Identity element~~ Operation must be associative

omputing.llnl.gov/tutorials/openMP
Intro to openMP - Tim Mattson

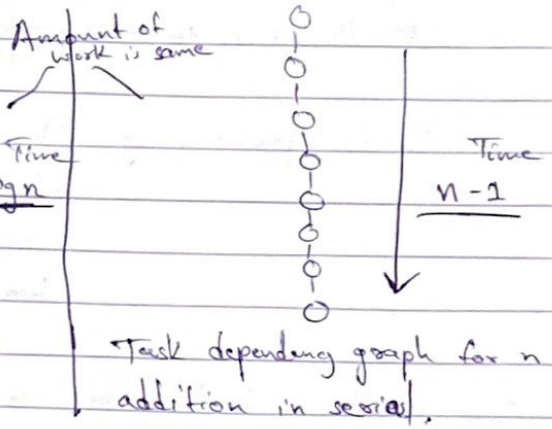
- Identity element must be present
- For reduction if it is shared as well as private.
  → ~~Opt~~ Best performance will be obtained.

* Module - 3 - Parallel Algorithmic Design

- Dividing ~~program~~ problem into task that can be executed concurrently/ in parallel → Decomposition

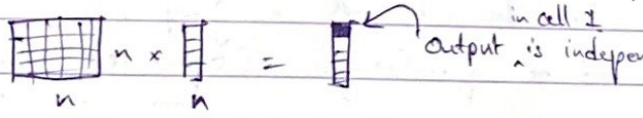


Task dependency graph for n additions in parallel



Task dependency graph for n addition in serial.

General Rule :- Map the output to the processor.



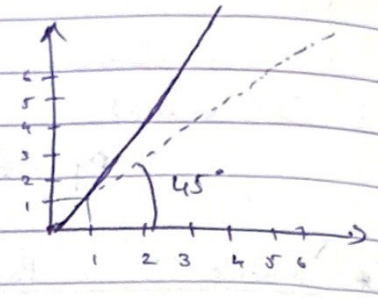
- For good design, control dependencies must be removed

getting together - Number of tasks that can be executed in parallel → degree of concurrency
It increases if granularity is low and vice - versa.

- Each node is the task, each edge is a dependency.
  Nodes should be more, edges should be less.

Note: Task dependency graph deals with control dependency.

- Superlinear Speedup



Not possible in theory.
But it is possible because as we introduce more processors, we introduce more amount of cache, thus it is possible to have superlinear speedup

Serial cost / ~~time complexity~~ : Order of time
Parallel cost / ~~time complexity~~ = Order of time × No of processors

To the previous eg. Work cost in serial = n
Work cost in parallel = $(\log_2 n)\, n$
Time complexity in serial = n
~~Work~~ Time complexity in parallel = $\log_2 N$

- Work = Total no of nodes / ~~task~~ / vertices if all tasks do same amt of work. else summation of work done by each
- Span / Depth (D) :- Longest path in graph.
- $\frac{W}{D}$ = Avg amt of parallelism. (Optimal number of of processor to keep all of them busy on average)
- $T_p(n) \geq D(n)$ | p is number of processor, T_p = Execution time | D is depth of graph

The minimum execution time is the depth of the graph

* Work Span Law : $T_p(n) \geq \max\left(D_n, \left\lceil \frac{W(n)}{p} \right\rceil\right)$

- Ratio of total work to critical path length. = Max degree of concurrency
= ?

IP Addrs of HPC Cluster
→ 10.100.71.130

- Critical path length (E_j) a → 27 , b →
- Each table - 10 time units.

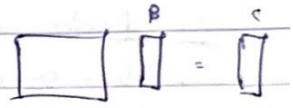
$W = 40 + 15 + 8 = 63$ ⌈7 / 3⌉ ≈ 3 processors.
27

* Task Interaction Graph

If the entire data is not replicated across all the tasks, they will have to communicate elements of the vector.

Types of Dependencies
- Data
- Loop
- Control
- Structural

} More imp



Eg. Multiplication of sparse matrix with vector
- Only non zero elements of A takes place in computation
- Matrix B is divided into partitions. Each stored distributedly

Task 0 has 1st row of A and B[0] and need to compute C[0]
Thus it will require B[1], B[4], B[8] as A[0][1], A[0][4] and A[0][8] is non zero.

Note Task Interaction Graph will be used to deal with data dependancies
- Helps in Load Balancing

Note As granularity increases (finer) the cost of overheads computation increases.

- Mappings are determined by both task dependency and task interaction graphs
- Task Dependency graph. → Load Balancing.
→ Optimise the use of processors

Lab Access Cluster through
SSH ID @ 10.100.71.130.
SSH ~~CSE0~~ GICS@

Work complexity

S	P
O(n)	O(n)

Step | Time complexity

S	P
O(n)	$O(\log_2 n)$

Cost complexity

S	P
n-1	$n \log_2 n$
O(n)	

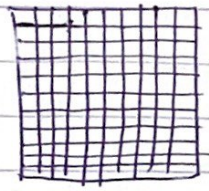
- Serial Implementation - Time Complexity of algo, If dependency or not.
- Implementation of approach - Can add mapping graph
- Complexity of parallel -
- Cost of parallel -
- Theoretical Speedup = $\frac{T_{serial}}{T_{parallel}} = \frac{n}{\log_2 n}$
- Estimated Serial Fraction — ✗
- Upper bound on Amdahl — ✗
- Number of memory Access - Inside code, put a counter, Find code by
- Number of computation - Can be found by counters.
- Time curve (Constant problem size, vary number of process)
- Time curve ((Constant processor, vary ~~number of~~ problem size)) -
- Speed up analysis.
- Efficiency curve ↦ Speedup / Number of processor (If actually dec
- Karp Flatt metric → ✗
- Further Detailed → ✗ (First - 3 Sections)

- Decomposition Techniques
  - Recursive :- Divide and Conquer → Decomposed to set of sub problems E
  - Imp - Data : :
    - Exploratory
    - Speculative

→ In recursive, the granularity increases
- In reduction, granularity decreases.

- Data Decomposition :- Partition the data on which computation are performed, across various tasks

- Input Decomposition → Output not known
- Output Decomposition → Each element of output can be calculated independantly of other task.
- Intermediate Decomposition -



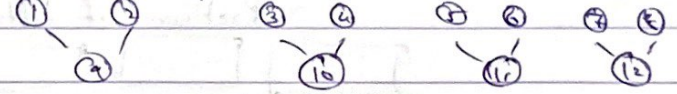
→ For it to be multiplied.
1st row with 1st column, 2nd column ...
But there is a possibility that cache will be filled and 1st few elements will be removed.

Thus if we have divided it then cache hit will be more...
Also work complexity remains the same. The size of block will be based on cache size.

- Work inefficient if work complexity (Parallel) > work complexity (Serial)
This can be valid even when step complexity (parallel) < step complexity (Serial).

- Intermediate decomposition of matrix multiplication

Task dependency →



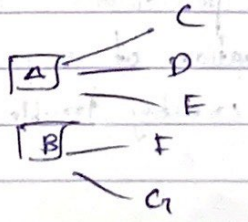
* Owner Computes Rule.
- Process assigned a particular data is responsible for all computation associated with it

(quicksort)

* Exploratory Decomposition
- Decomposition goes hand-in-hand with its execution.

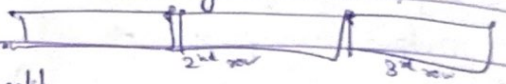
Eg - Game playing etc → Non deterministic systems.

* Speculative Decomposition :- Assumes that one of them would be correct



Starts processing C as it speculates that C will be execute

- For 2-D array - If it is stored in memory as flat



Thus there is problem of spatial locality.

In Lab → Check the schedule directives

- Scan Prefix sum

Given array - $A = \{a_0, a_1 \ldots a_n\}$

$Scan(A) = \{I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2\}$

This is difficult to parallelise → Each element depends on its previous

Scan does $\log n$ parallel iterations

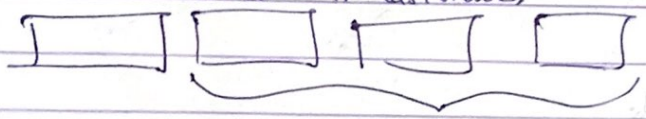
$(n-1) + (n-2) \ldots$ $\log n$ terms

$\Rightarrow n \log n - (n-1)$ $\approx O(n \log n)$ Complexity

Thus work complexity is more than serial $\Rightarrow$ Work inefficient.

Work efficient → Build balanced binary tree → Blelloch

→ Problem if distributed



These depends on the previous → becomes serial

Scan can be used in compaction.

1	2	3	4	5	6	7
0	1	0	0	1	1	0
Scan 0	1	1	1	2	3	3

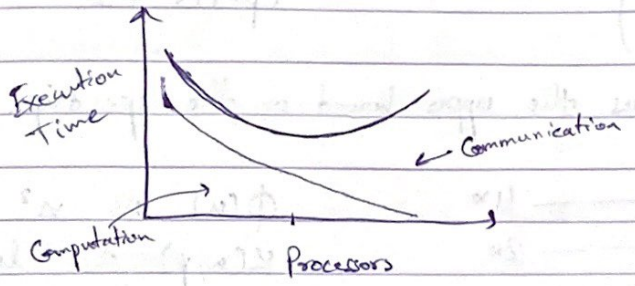
2	5	6	1	3	4	7

Lab

1) Amdahl's Law → possibility of parallelization
2) Gustafsson - Barcis Law - Evaluation of a parallel program
3) Karp - Flatt Metric - Which is creating trouble ? Serial or Overhead
4) Isoefficiency Metric.

- Sequential Time - $\sigma(n)$ — $n \rightarrow$ problem size
- Parallel Time - $\phi(n)$ — $p \rightarrow$ processor
- Overhead → $K(n,p)$
- Speedup → $\psi(n,p) = \frac{T_s}{T_p} \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + K(n,p)}$

- If p increases Communication time increases
  Computation time decreases



Thus for a given problem size there is an optimal number of processors.

Efficiency = $\left(\frac{T_s}{T_p}\right) \times \frac{1}{p}$

↓

Amount of processor utilization

$0 \leq \epsilon \leq 1$

$E(n,p) = \frac{1}{p}\left(\frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + K(n,p)}\right)$

Thus efficiency $\epsilon \leq \frac{\sigma(n) + \phi(n)}{p(\sigma(n) + K(n,p) + \phi(n))}$

Thus if $\sigma(n) = 0$ and $K(n,p) = 0$ Then. $\epsilon \approx 1$.
if $\phi(n) = 0$ then $\epsilon = \frac{\sigma(n)}{p(\sigma(n) + K(n,p))}$.

Thus $\epsilon \rightarrow 0$ when $p \rightarrow \infty$

- **Amdahl's Law** :- $\psi(n,p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + K(n,p)}$

Ignoring overheads

$\psi(n,p) << \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p}}$

Let $f = \frac{\sigma(n)}{\sigma(n) + \phi(n)}$ → Fraction of serial

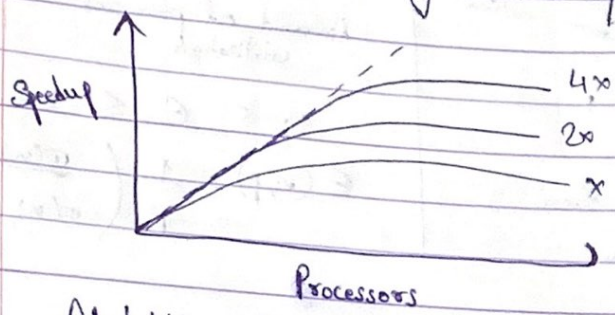
$$\therefore \psi(n,p) = \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p}} = \frac{1 + \phi(n)/\sigma(n)}{1 + \phi(n)/p\sigma(n)}$$

$$\frac{f}{1-f} = \frac{\sigma(n)}{\phi(n)} \quad \therefore \psi(n,p) = 1 + \left(\frac{1-f}{f}\right) \Big/ 1 + \left(\frac{1-f}{pf}\right)$$

$$= \frac{1}{f} \Big/ \frac{pf + 1 - f}{pf}$$

$$\therefore \psi(n,p) = \frac{1}{f + \left(\frac{1-f}{p}\right)} = \frac{p}{(p-1)f + 1}$$

Thus Amdahl's law gives the upper bound on the speedup.



$\phi(n) \approx n^2$

$K(n,p) \approx \log n$

Amdahl's effect :- For fixed number of processors, the speedup increases as the problem size increases.

- Gustaffson - Barsis law :- Time is fixed

Let s = Fraction of time spent in the parallel code performing serial work

$$s = \frac{\sigma(n)}{\sigma(n) + \frac{\phi(n)}{p}} \qquad 1 - s = \frac{\phi(n)/p}{\sigma(n) + \phi(n)/p}$$

$$\sigma(n) = \left(\sigma(n) + \frac{\phi(n)}{p}\right) s$$

$$\phi(n) = \left(\sigma(n) + \frac{\phi(n)}{p}\right)(1-s)\,p$$

$$\psi(n,p) \leq \frac{\left(\sigma(n) + \frac{\phi(n)}{p}\right)(s + p - sp)}{\sigma(n) + \frac{\phi(n)}{p}}$$

$$\therefore \quad \psi(n,p) \leq s + p(1-s)$$
$$\leq p + (1-p)s$$

- Overheads in parallel code are due to the following
  - Creation of threads - Process Setup
  - Communication
  - Extra Computation.
  - Coordination and Synchronization.

- Experimentally determined Serial fraction

$$T(n,p) = \sigma(n) + \frac{\phi(n)}{p} + K(n,p)$$

$$T(n,1) = \sigma(n) + \phi(n) \quad \rightarrow \text{Serial}$$

Experimentally determined Serial fraction

$$e = \frac{\text{idle time} + \text{Overhead}}{}$$

$$= \frac{(p-1)\sigma(n) + pK(n,p)}{(p-1)\,T(n,1)}$$

idle time $= (p-1)\sigma(n)$

Tot Overhead $= pK(n,p)$

$$= \frac{p(\sigma(n) + pK(n,p) - \phi(n)}{(p-1)\,(T(n,1))}$$

$$= \frac{pT(n,p) - \phi(n) - \sigma(n)}{(p-1)\,T(n,p)}$$

$$= \frac{pT(n,p) - T(n,1)}{(p-1)\,T(n,p)}$$

$$\Rightarrow \quad pT(n,p) = e\,[(p-1)\,T(n,1)] + T(n,1)$$

$$\Rightarrow \quad pT(n,p) = epT(n,1) + T(n,1) - eT(n,1)$$

$$\Rightarrow \quad T(n,p) = T(n,1)\,e + T(n,1)\left[\frac{1-e}{p}\right]$$

$$\Rightarrow \quad T(n,p) = T(n,p)\,\psi\,e + T(n,p)\,\psi\left(\frac{1-e}{p}\right)$$

As $\psi = \dfrac{T(n,1)}{T(n,p)}$

$$1 = \psi e + \psi\left(\frac{1-e}{p}\right)$$

$$\Rightarrow \frac{1}{\psi} = e + \frac{1-e}{p} \Rightarrow \frac{p}{\psi} = (p-1)e + 1$$

$$\Rightarrow \left(\frac{p-\psi}{\psi}\right)\frac{1}{p-1} = e$$

$$\Rightarrow \boxed{e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}}$$

-

p	2	3	4	5		8
ψ	1.82	2.5	3.08	3.57		4.71
e	0.1	0.1	0.1	0.1		0.1

Since the value of e is constant, there is no overhead

⇒ Problem with serial → May be large computations need to be done in se

-

p	2	3	4	5		8
ψ	1.87	2.61	3.2	3.73		4.71
e	0.07	0.075	0.08	0.085		0.1

Since the value of e increases, there is large overhead Thus there is scope of improvement.

- For a given fixed problem size, the efficiency decreases as you increase the number of processors

- **Isoefficiency Metric** - $\psi(n,p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + K(n,p)}$

Measure of the ability to increase the performance as the number of processors increases.

$$\psi(n,p) \leq \frac{(\sigma(n) + \phi(n))\,p}{\sigma(n) + \phi(n) + (p-1)\sigma(n) + pK(n,p)}$$

Overhead = $T_o(n,p) = (p-1)\sigma(n) + pK(n,p)$

Thus efficiency $\epsilon(n,p) = \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) + T_o(n,p)}$

$$\epsilon(n,p) \leq \frac{1}{1 + \frac{T_o(n,p)}{\sigma(n) + \phi(n)}} = \frac{1}{1 + \frac{T_o(n,p)}{T_o(n,1)}}$$

$$\Rightarrow \frac{T_o(n,p)}{T(n,1)} \leq \frac{1 - \epsilon(n,p)}{\epsilon(n,p)}$$

$$\Rightarrow \boxed{T(n,1) \geq \left[\frac{\epsilon(n,p)}{1 - \epsilon(n,p)}\right] T_o(n,p)}$$

Thus It tells you the rate at which the problem size must increase such that efficiency remains constant.

Block Matrix Multiplication (Can use part to calculate cache hits)

Let cache hit time = 1 cycle

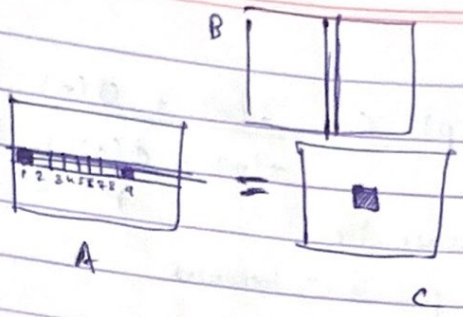
Miss penalty ≈ 100 cycle

Average Access time for 99% = $\frac{99}{100} \times 1 + \frac{1}{100} \times 100 = 1.99$ cycles

Avg access time for 97% = $\frac{97}{100} \times 1 + \frac{3}{100} \times 100 = 3.97$ cycles

⇒ Takes almost twice the value.

Of Avg access time for 80% = $\frac{80}{100} + \frac{20}{100} \times 100 = 20.8$ cycles



L_1 - 32 Kb
L_2 - 256 Kb
L_3 - ~~64 Mb~~ 6-8 MB

Matrix elements = 8 bytes
Total operations = $O(N^2)$
Cache block size = 64 bytes
Cache Miss = $\frac{N}{8}$ (For row) + N ← For column. = $\frac{9N}{8}$

This is for 1 output → Thus Cache Miss $O\left(N^2 \cdot \frac{9N}{8}\right)$

- Using Blocks - Block size = B^2
  The cache can hold 3 Blocks ($3B^2 < C$)

- Cache Miss for each block = $(B^2/8) \cdot \frac{2N}{B}$

∴ Total Cache miss = $\frac{2N}{B} \times \frac{B^2}{8} = \frac{nB}{4}$

Total Blocks = $\left(\frac{N}{B}\right)^2$

∴ Total Cache miss = $\left(\frac{n}{B}\right)^2 \times \left(\frac{nB}{4}\right) = \frac{n^3}{4B}$

- General observation - Keep your working data set very small
  Keep stride sizes small. (jumps in memory)