High Performance Computing (HPC)

~~Open MP~~ : Shared Memory
Distributed

C, OpenMP, OpenMPI → Technical

www.letshpc.org

Performance → operations / computations per unit time

Algorithms		Computer
1) Deterministic		Non-deterministics
2) Abstract		
3) Terminating		Non terminating

System / Architecture make deterministic → non determ

Agenda → Accuracy + Efficiency

Performance
↓ unit
flops (floating point operations/ ~~processor~~)
flops / sec

- Frequency of the processor (cycles/sec)
- Number of cores
- Flops / cycle

Clocks/s Flops/cycle No. of cores

$\frac{2.5 \times 10^9}{\text{second}} \times 4 \times$

$= 10$ G Flops / sec

Multiply it with the number of cores.

Serial programming → 10 G Flop/sec.

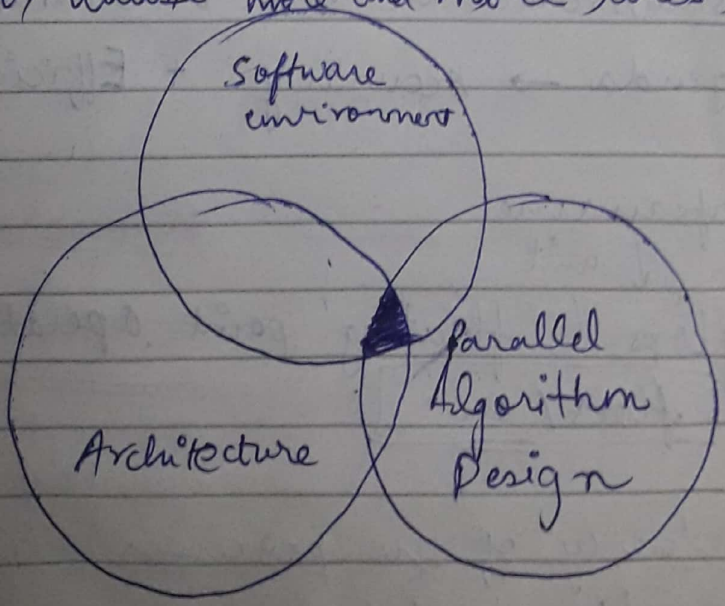
Performances

Architecture (Parallel) Algorithm. Software environment

Cache memory

① Portability ② Scalability

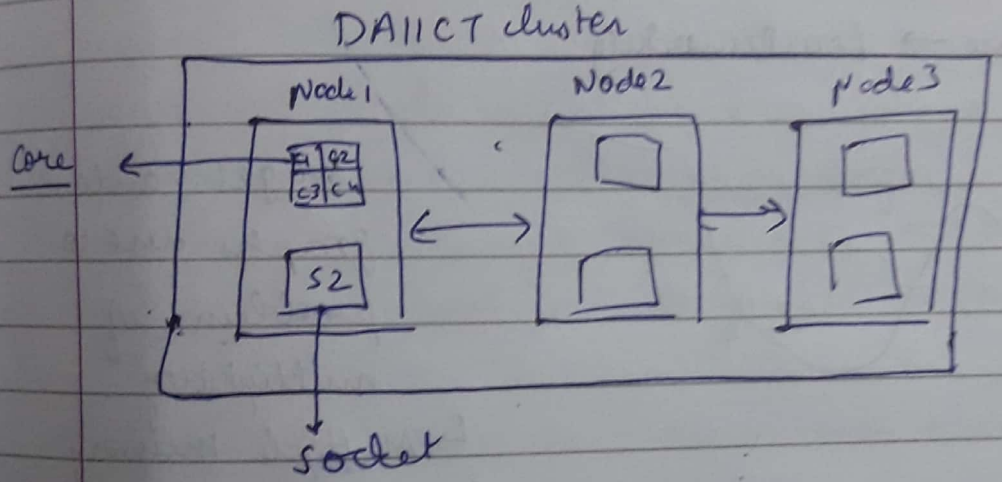
able to utilise more and more cores;



computation → operation + data

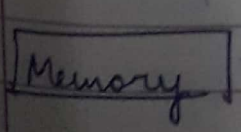
Architecture → ND (Non Deterministic)
Algorithm Design → D
Software environment → ND

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



Supercomputer = several nodes → HPC

Node performance in Gflops = (CPU speed in GHz) × (number of CPU cores) × (CPU instruction per cycle) × (number of CPUs per node)



① Bandwidth → MB/s
② Latency → seconds

von Neumann bottleneck

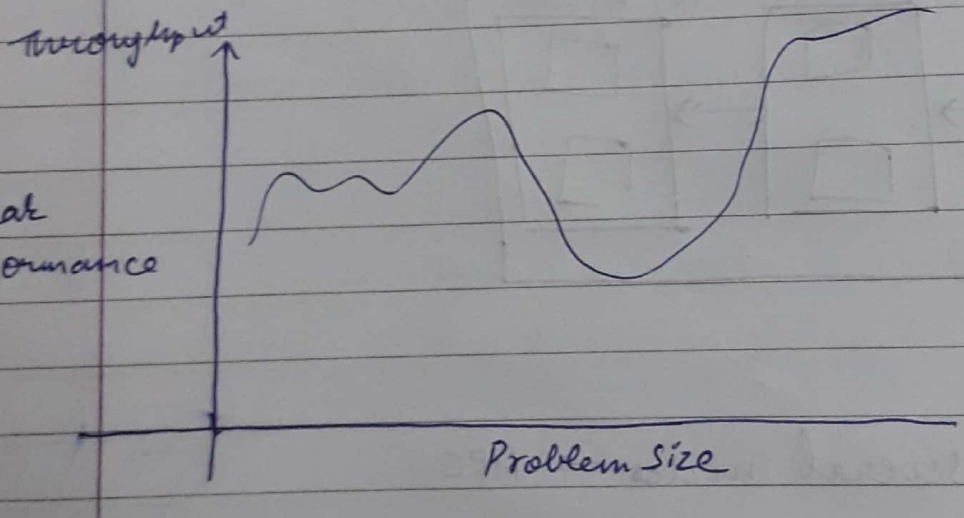
SISD (Single Instruction Single Data)
SIMD MISD MIMD

- Instruction Stream
- Data Stream

Task.

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

Peak performance → benchmarking



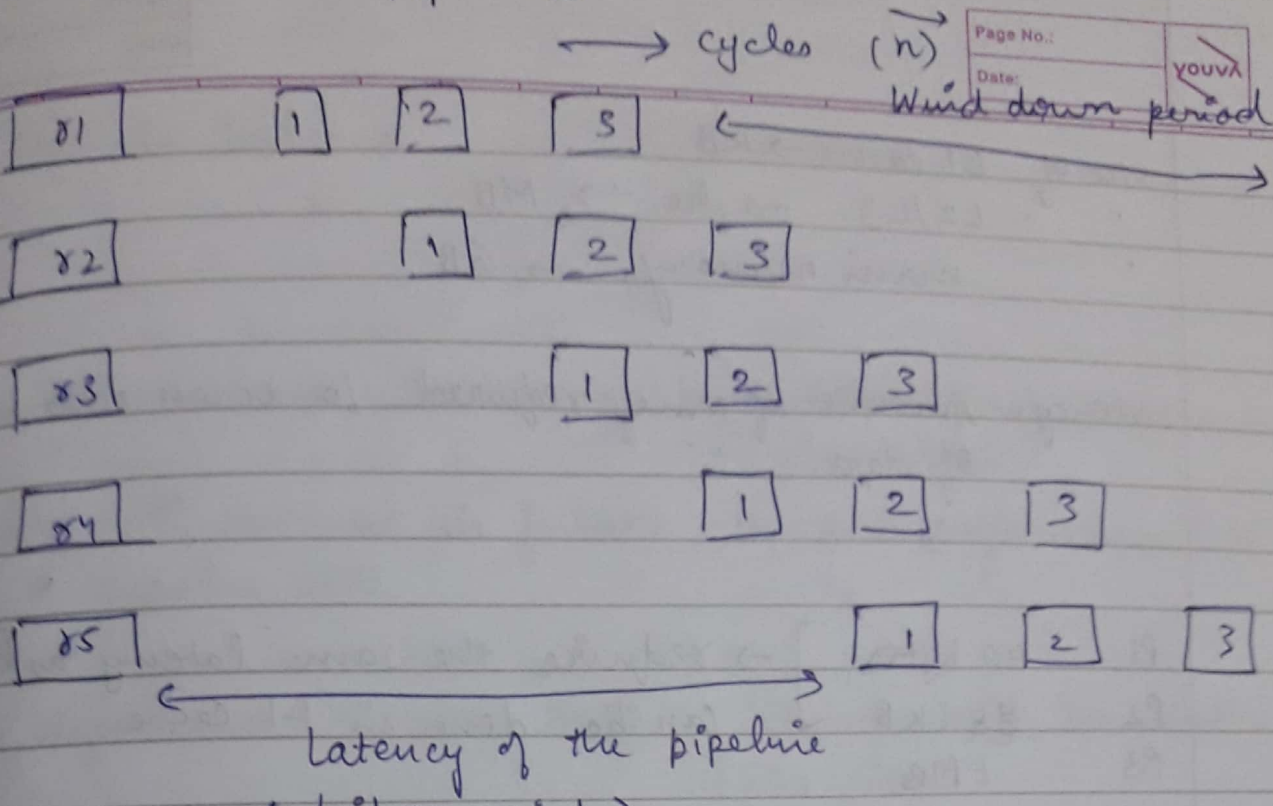
We get random graphs due to problem of multiplicity.
↳ Multiple memory
↳ Multiple data path
↳ Multiple process

Our aim is to get best performance

Parallelism possible in various level.

* Node * Thread * Instruction Level (Pipelines)

Pipelines



- 1 result / cycle after wind up phase
- Depth of the pipeline = m stages.
- Total Independent operations → N

Time of pipeline to finish → (N + m - 1)

Serial (no pipeline) time → Nm

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

$N \gg m$

we get much higher speedup.

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

For higher values of N ⇒ Throughput ≈ 1

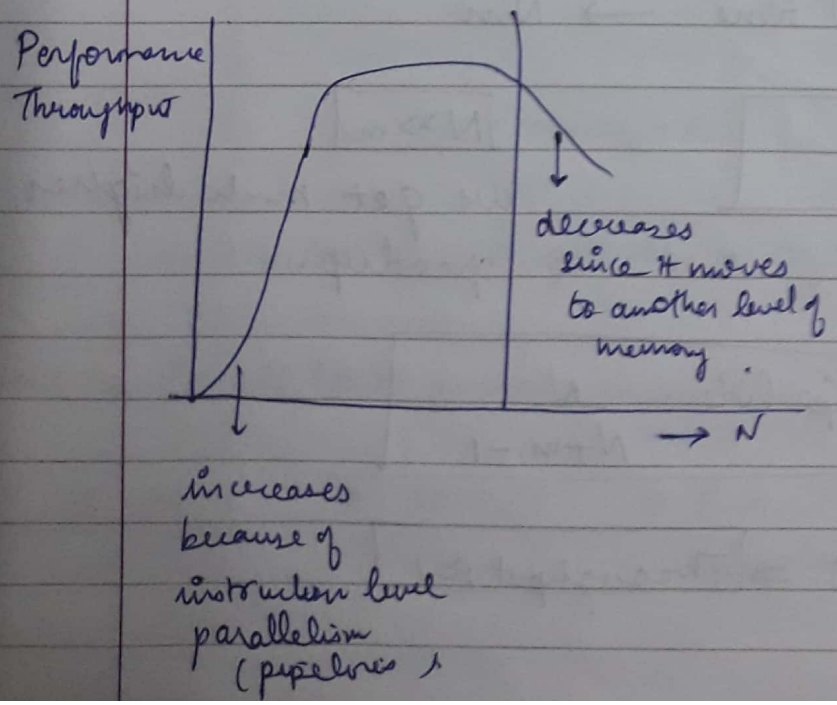
Size of L1 cache → KB
" " L2/L3 cache → MB
" " main memory → GB

Latency: Amount of time required to access zero bytes of data.

P1 40 bytes
P2 8B 1KB
P3 1MB
] → require the same latency as both can be done in L1 cache.

Performance of CPU is increasing at a much greater rate than performance of memory (DRAM) [Von Neumann bottleneck].

→ Parallel vector triad benchmark.



Machine balance | Code balance

Machine balance = memory bandwidth / peak performance
(bytes/flops)

It will decrease in future as peak performance ↑ at a greater rate

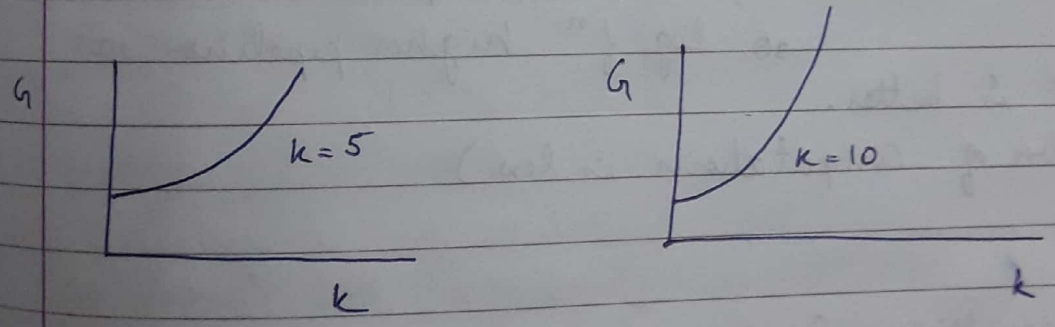
Code balance = data traffic / floating point operations
(bytes/flops)

Code balance of vector triad = 4/2 = 2 bytes/flops.

Reciprocal of code balance → computational intensity

∴ Lower code balance is desirable

Gain $G = \frac{k}{\gamma + (1-\gamma)k}$ $\quad 1 \le G \le k$



2 factors: speed and size

→ Cache line length = 16 ⇒ Hit ratio = 15/16
(Spatial locality)

(1st A, cannot hit, not present on cache, initially A [1:n] bring 16 to cache, I can't hit
15/16 hit ratio

Spatial + Temporal locality
→ (2 terms related to

Cache is k times faster than RAM
↳ (Latency + Bandwidth)

Cache reuse ratio is γ (fraction)
Access to main memory (t_m)

Cache access time = $\frac{T_m}{k} = T_c$

Average access $T_{av} = \frac{\gamma \times T_m}{k} + (1-\gamma) T_m$

$= \gamma T_c + (1-\gamma) T_m$

Performance gain = $\frac{T_m}{T_{av}} = \frac{T_m}{\gamma T_c + (1-\gamma) T_m} = \frac{k T_c}{\gamma T_c + (1-\gamma) k T_c}$

Case A
Case B

A = A + B**2.0 (case A) A, B are float
A = A + B*B (case B)

B is better as (B**2.0) ↔ exp(2 ln B)
so log f^n higher pipeline so
(case B) is better.
(Number of computations is less)

Avoid conditional branching:

- Since in conditional branching there is a possibility of branch miss
- compiler ~~has to~~ makes a guess and if it is proved wrong he has to flush the pipeline.

Loop Unrolling

Chances of cache hit increases.

Organisation of Parallel Platforms

- Logical
  - Programmer View
    - Expressing Parallel Tasks
      - Several Models (control Structure)
    - Mechanism for Communication
- Physical
  - Architecture

Granularity: Ratio of computation to communication.

Coarse: large computational work b/w communication
Fine: small

Matrix Multiplication

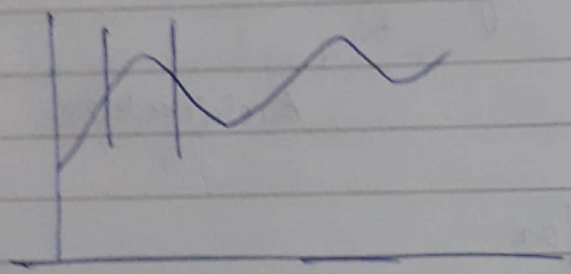
Finest level of granularity = N^2
Coarse " " " " : $\frac{N}{4}$

Open MP (Multi Processing → MP) (Shared systems)
Open MPI (Message passing) (Distributed systems)

start

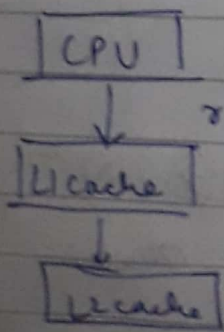
* not necessary to compile using -fopenmp
(How to measure one clock time)

* Resolution of the clock.
~~Eda~~ (We need to take multiple measurements



Tight loops ≈ ~~too~~ length of the loop is so small
(comparable with the stages of pipeline

Loop Size → 10^3
4 arrays → 4×10^3
(Double precision) → 32×10^3 = 32K (Maximum ~~to~~ size of L1 cache)



latency of L2 cache more than L1 cache.

Memory System

64 bit DDR AM interface
8 channels
1 GHz clock frequency

(DDR) → Double Data Rate

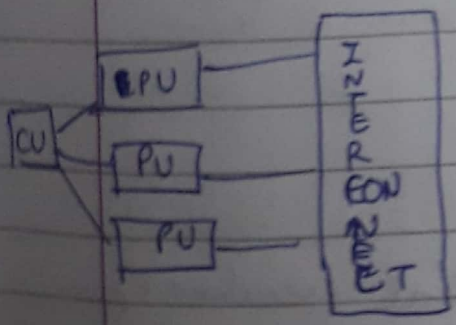
Access latency of 200 cycles

→ Access Throughput (bytes / sec) = $\frac{8 \text{ bytes}}{\text{tranfer}} \times 8 \text{ channels} \times 2 \text{ (DDR) } \frac{\text{tranfer}}{\text{channel} * \text{(clock / cycle)}} \times 10^9 \frac{\text{clock}}{\text{second}}$

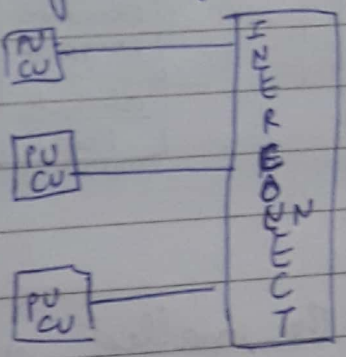
= 128 Gbps

L3 cache is a shared cache (multiple threads can access)

SIMD
(Single Instruction Multiple Data)

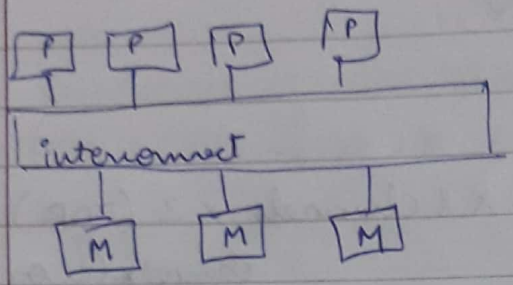


SPMD
(Single Program Multiple Data)

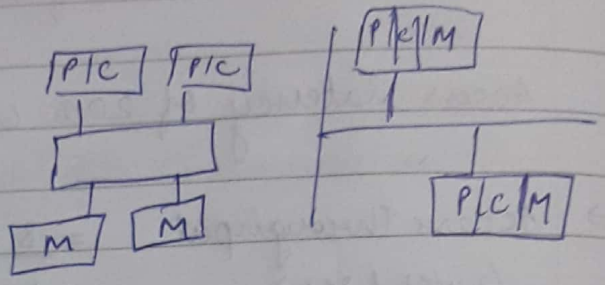


Memory : Private
Shared

UMA: Uniform Memory Access
NUMA: Non uniform Memory Access.



UMA



NUMA

(Each process can access its own cache very fast however cannot access other cache quickly)

- context switching (diff. from parallel programming)

Process	Thread
• Running program	• Light weight Process
① PID ② status	→ stack
Stack Executable program Data Text Heap	Executable program Global Heap

* Thread is the smallest unit of sequential execution of instructions.
* Process is an execution stream

Status of the process:
- New • Running • Ready • Waiting • Terminated

- Threads have independent stacks but share heap and global variables.

Thread includes (individual)

* Thread ID * Program Counter * Register Stack * Stack
(pointer) (pointer)

(shared)

* Text section * Data section * other OS resources

OS uses process control block

* execution state for each thread
* scheduling information
* Memory used by the process
* Information about open files

(S)MD ~~too~~ ~~more~~ takes ~~more~~ time in scheduling as it has only

(SIMD takes less time in scheduling as it has only a single CU whereas SPMD takes more time in scheduling)

(~~who~~ Every thread can read at the same time whereas while writing only one thread can write at a time)

Dependencies:

```
for i = 1 . . n
    for j = 1 . . . m
        b[i, j] = . . . + b[i, j-1]  → Dependency on j
```

4 kinds of dependencies:
① Flow, ② Anti ③ Input ④ Output

→ Flow (true dependence)
Statement 'i' precedes 'j'
and 'i' computes a value that 'j' uses

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

① → ②
② → ④ [These are the dependencies]

→ Anti dependence
Statement 'i' precedes 'j'
and 'i' uses a value that 'j' computes

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

② → ③
② → ④

→ Output dependence

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

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

① → ③
① → ④
③ → ④

→ Input dependence

Statement 'i' precedes 'j'
and 'i' uses a value that 'j' also uses

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

S → T

* Difficult to remove flow dependence.

→ Dependence flows from i to j
i → source
j → sink

Two important dependencies:

→ Data dependency → Task dependency
(represented using DAGs)

Data dependency graph

- Nodes are the statement
- Edges are the dependencies relations

** loop carried dependence

a[i] = b[i] + c[i] → (Flow depend
d[i] = a[i] [loop independent

loop independent depend

a[i] = b[i] + c[i] → (loop carried dependences)
d[i] = a[i-1]

loop distribution
loop fusion

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

Can be parallelised
locality is worse
(more overhead due to parallel processing)

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

cannot be parallelised.
locality is better
(spatial locality)

→ Loop fusion done for better utilisation of spacial locality
→ Loop distribution to bring parallelism.

directives : Special preprocessor instructions

Shared : all processors have a shared memory
→ bus contention leads to programs scalability

Distributed : all processors have distributed memory
→ memory management is difficult

Fork-Join concept

→ Fork creates concurrency
→ Join removes concurrency
→ Fork children can join with the parent in any order.

OpenMP → Structured blocks : One point of entry and exit

(Inside structure blocks if condition is not allowed)

Scope → Access
→ & Time it is valid

Page No.:
Date: YOUVA

- Global variables shared among threads
- Private variables
  - → exist only within the new scope; they are uninitialised and undefined outside the scope
  - → loop variables

Create Parallel Regions

C

```
# pragma omp parallel
  {
     code to be executed by each thread
  }
```

- fopenmp.

- OMP_NUM_THREADS // determines how many threads are initialised

```
export OMP_NUM_THREADS = 4
setenv      "      4
```

~~#include "omp.h"~~
~~void main()~~

SPMD

↳ (decreases concurrency)

```
# pragma omp critical // only one thread can execute
  global_result += my_result;
```

pragma omp parallel → this directive creates threads

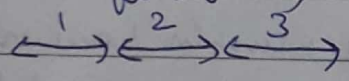
pragma omp for → this directive does work sharing

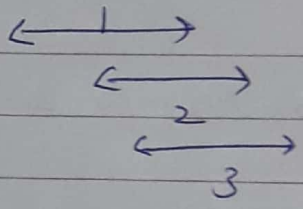
Three ways:

① Define a variable i explicitly inside parallel
② # pragma omp for (use inside this)
③

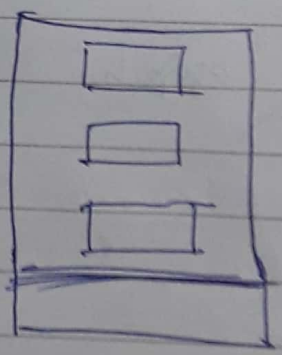
Index variable of a worksharing loop is private by default.

	Synchronous	Asynchronous
→	~~multiple threads at same time~~ only one thread at a time (sequential)	→ can have multiple threads at one time
→	(Blocking Mechanism)	→ (Non blocking mechanism)





Data scope
Coordination
Synchronisation
Load balancing
Work sharing
Scheduling



Barrier

(a) event synchronisation
(b) explicit synchronisation
↳

⚠ By defining number of threads we control granularity

Page No.:
Date:
YOUVA

Floating point operations are not associative.
(due to round-off error)
While using reduction there should be an identity element.

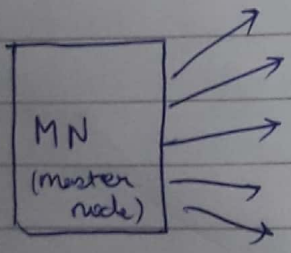
≠

Types of dependencies we have seen :

- control · data · loop · structured

Task dependency graph → control dependency
Task interaction graph → data dependency

IP of HPC cluster : 10.100.71.130



4 compute nodes
(open MP)
MPI

Any directory created on any node will be created in the rest of the nodes

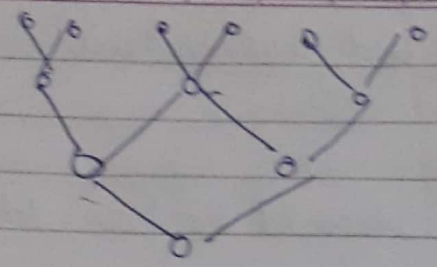
SSH ID @ 10.100.71.130

Top

SSH GICS0
GICS1
GICS2

- Write all entries in an editor first and then post on Lets HPC

→ Serial : Time implementation, whether dependency or not.



- Work
  serial
  $O(n)$
- Time / State
  $O(n)$
- Cost
  $O(n)$

- Complexity in parallel
  $O(n)$
  complexity
  $O(\log_2 n)$
  complexity
  $O(n \log_2 n)$

Theoretical speedup $= \frac{T_s}{T_p} = \frac{n}{\log_2 n}$

→ Ignore estimated serial fraction and upper bound and Karp-Flatt metric analysis.

Number of memory access and computations.

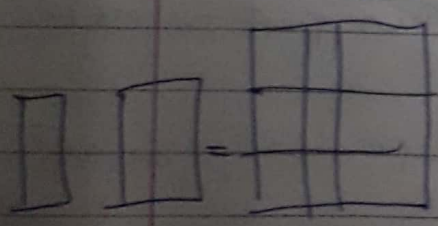
Speedup (n , p)
↓ ↓
Problem size, Number of processing.

Efficiency = $\frac{\text{Speed up}}{\text{Processor}}$

=> Recursive Decomposition → Quicksort
(Granularity is increasing) →

→ Data Decomposition

Output data decomposition:



Output data decomposition

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$C_{11} = A_{11}B_1 + A_{12}B_{21}$

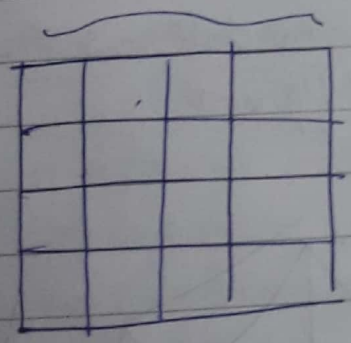
$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

We can further divide the task further but the concurrency will be equal.

More concurrency will be achieved but more memory will be required.

Block Matrix Multiplication :

n 10000



$10000 \times 8 = 80{,}000$ bytes = 80 kB

Work inefficient parallel algorithm ;

if work complexity (P) > work complexity (S)

but step complexity (P) < step complexity (S)

can

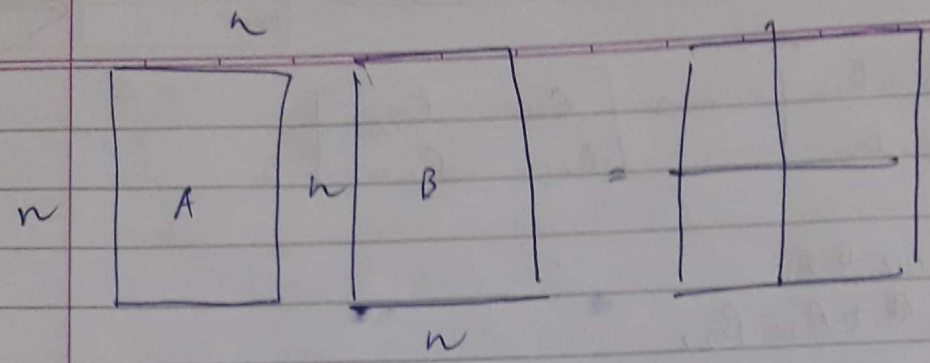
- Decide the size of the block based on your cache size

Intermediate data decomposition

Divide the task into 8 parts and combine it again

$D_{11} = A_{11}B_1$

$D_{12} = A_{12}B_{21}$



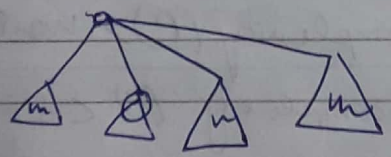
If one processor assigned a data then associated with all the computations associated with it.

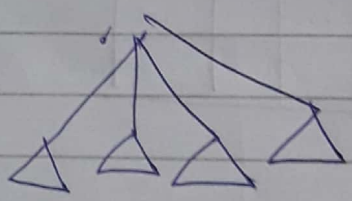
Sorting an array : input data decomposition.

Exploratory decomposition:

~~Based on~~ Since the program is non deterministic
work complexity will be different.

Anomaly





Serial work: $2m+1$
Parallel work : 1

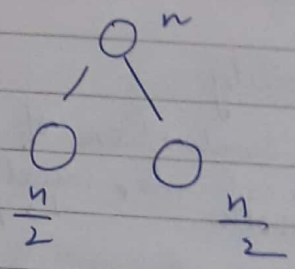
Serial work: m
Parallel work: $4m$

Speculative Decomposition : Don't know where to go next

Hybrid decomposition : mix of data and recursive decomposition.

Commands

screen top ps & ssh scp



$l = 2$

$\frac{n}{b^x} = 1$

$\log n = x \log b$

$x = \frac{\log n}{\log b} = \log_b n$

Step complexity : $\log_b n$

Work complexity : sum of all steps.

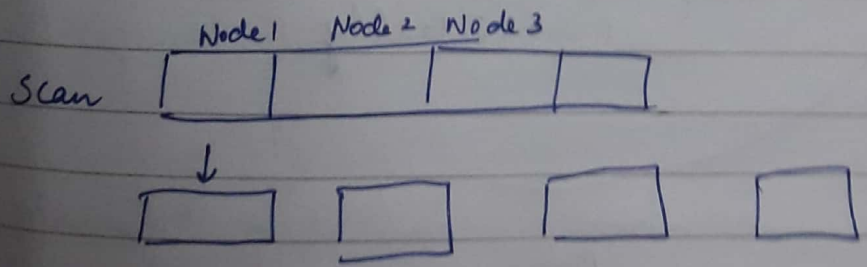
Scan Algorithm

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

$n \log n - (1 + 2 + 4 + \ldots)$

$n \log n - (2n - 1)$

$= O(n \log n)$



Performance Analysis

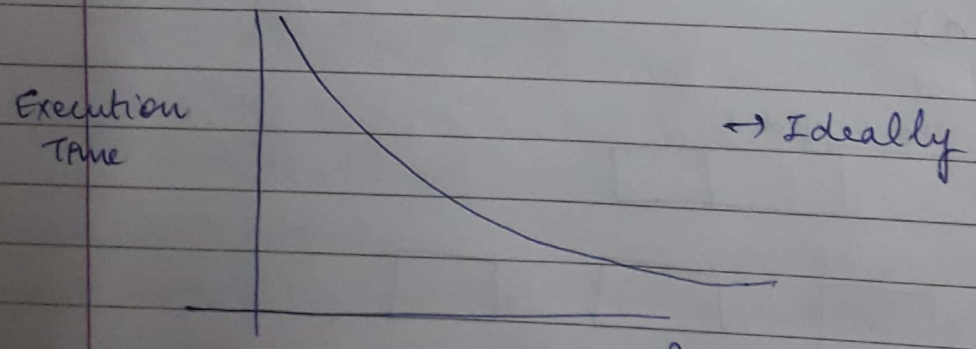
1) Amdahl's Law → possibility
2) Gustafson - Barsis Law → evaluation of a parallel program
3) Karp - Flatt Metric → Overheads, serial, parallel → problem size
   ↳ both ↳ problem size
4) ISO - efficiency analysis.
   ↳ scalability

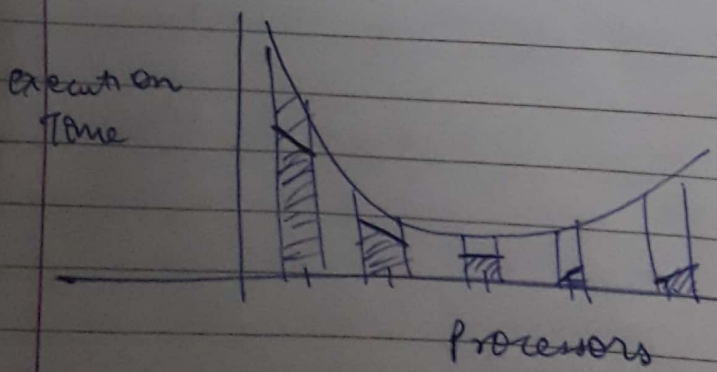
Problem size → n
P → processors

Sequential Time → $\sigma(n)$
Parallel Time → $\phi(n)$ [Total time]
Overhead → $K(n,p)$

[Speedup] $$\Psi(n,p) = \frac{T_s}{T_p} \leq \frac{\sigma(n) + \Psi(n)}{\sigma(n) + \frac{\Psi(n)}{P} + K(n,p)}$$

P ↑ = Communication Time ↑ + Computation Time ↓





▨ → computation
□ → communication

fixed problem size

for a given problem size there is an optimal number of processors that will give you.

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

$0 \le E \le 1$ (n,p)

Amount of processor utilisation.

$$E(n,p) \le \left(\frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+k(n,p)}\right)\frac{1}{p}$$

$$= \le \frac{\sigma(n)+\varphi(n)}{p\sigma(n)+\varphi(n)+pk(n,p)}$$

In an ideal code, $\sigma \to 0$, $k \to 0$

$$E(n,p) \le 1$$

In worst case

$\varphi(n) \to 0$ and $p \to \infty$

$$\ge \lim \frac{\sigma(n)}{p(\sigma(n)+kn(p)} = 0$$

$\therefore 0 \le E(n,p) \le 1$

Amdahl's Law:

$$\psi_{(n,p)} \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+k(n,p)}$$

$$\psi_{(n,p)} \ll \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p}$$

Derive Amdahl's Law [not very practical as not considering overhead]

* Fixed problem size

$$f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)} \rightarrow \text{serial part}$$

$$\psi(n,p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p}} = \sigma(n) f$$

$$\varphi(n) = \frac{\sigma(n) - f\,\sigma(n)}{f} = \frac{\sigma(n)}{f}(1-f) = \sigma(n)\left(\frac{1}{f} - 1\right)$$

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

$$\boxed{\psi(n,p) \leq \frac{1}{f + \frac{1-f}{p}}}$$

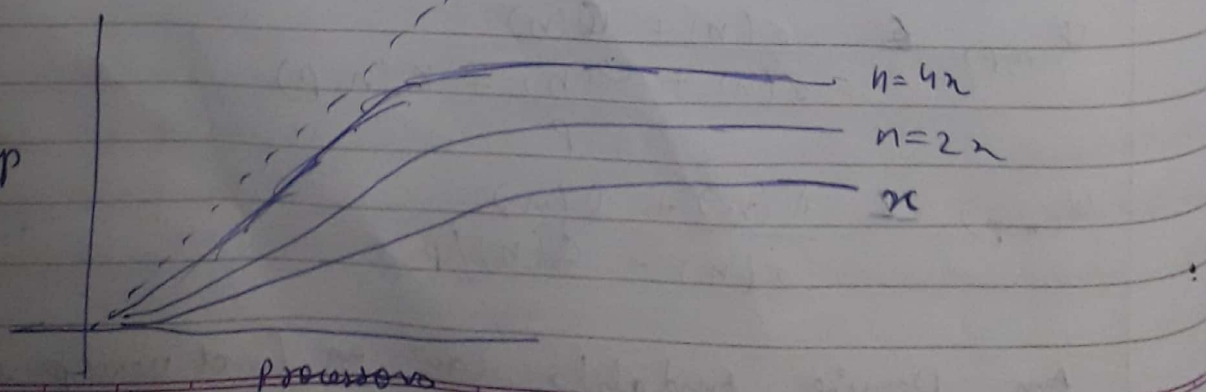
If $f \uparrow$ then poor speedup

If $f \downarrow$ then good speedup

* However bad analysis since not considered overhead

- Amdahl's law gives us an upper bound on speedup with a fixed problem size and neglecting overhead.

$k(n,p)$ grows slower than $\varphi(n)$ for the problem size



For a fixed number of processors speedup increases as problem size increases (Amdahl's effect)

→ Do serial matrix mul/s block matrix mul.

2 ways to do BMM:

$$\psi(n,p) \le \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + k(n,p)}$$

Gustafson-Barsis Law : //Here two we neglect the overhead

S → fraction of time spent in the ~~serial~~ parallel code performing serial work.

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

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

$$\boxed{\sigma(n) = \frac{S\phi(n)}{p(1-S)}}$$

$$\psi(n,p) \le \frac{\left(\sigma(n) + \frac{\phi(n)}{p}\right)(S + (1-S)p)}{\sigma(n) + \frac{\phi(n)}{p}}$$

$$\boxed{\psi(n,p) \le p + (1-p)S}$$

// The speedup we can get by putting p number of processors performance.

Karp Flatt Metric

Overhead:

- ↳ creation of threads. → process set up.
- ↳ communication
- ↳ extra computation
- ↳ synchronisation.

Experimentally determined serial fraction.

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

→ $T(n,1) = \sigma(n) + \phi(n)$ → serial

~~e = Total amount of ideal~~ (idle, overhead)

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

$$p\,\sigma(n) + p\,\phi(n) - \sigma(n) + p(K(n,p)$$
$$= pT(n,p) - \phi(n) - \sigma(n) = pT(n,p) - T(n,1)$$

$$\boxed{pT(n,p) = e(p-1)T(n,1) + T(n,1)}$$

$$pT(n,p) = T(n,1)\,ep - T(n,1)e + eT(n,1)$$
$$T(n,p) = T(n,1)e + T(n,1)\left(1 - \frac{e}{p}\right)$$

$$\boxed{T(n,p) = T(n,p)\,\psi\, e + T(n,p\,\psi\, \frac{(1-e)}{p}}$$

$\frac{\frac{1}{3} - \frac{2}{5}}{\frac{2}{3}} =$

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

$\frac{1}{4} - \frac{1}{4} = 0.99$

Page No.:
Date:
YOUVA

$$1 = \psi e + \psi(1-e)/p$$

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

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

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

→ ~~Ideal time remains same when~~ e is constant
Overhead is zero
And we cannot improve on our code.
Problem in the serial part.

p	2	3	4	5	....	8
ψ	1.87	2.61	3.2	3.73		4.71
e	0.07	0.075	.08	.085		.7

When the overhead is very high

Isoefficiency Metric

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

Ability to increase performance as number of processors increase.

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

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

$$E(n,p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) + T_o(n,p)}$$

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

$$\frac{T_o(n,p)}{T_o(n,1)} \leq \frac{1 - E(n,p)}{E(n,p)}$$

$$T(n,1) \geq \frac{E(n,p)}{1 - E(n,p)}\, T_o(n,p)$$

Rate at which the problem size has to be increased so that efficiency remains constant.

Slope of LHS much greater than RHS

Bandwidth	Latency (ns)
GB/s	L_1
	L_2
	RAM

Latency ↑ increases by a factor of 10 at every step

Cache hit ~~rate~~ time = 1 cycle 99% hit rate

miss penalty = 100 cycle

Average access time = $\frac{99}{100} \times 1 + \frac{1}{100} \times 100 = \underline{\underline{1.99}} \approx 2$

~~97%~~ $\frac{97 \times 1}{100} + \frac{3 \times 100}{100} = \frac{397}{100}$ 97% hit rate $= 3.97 \approx 4$

(2 times slower than 99% hit rate)

80% hit rate

$\frac{80}{100} \times 1 + \frac{20}{100} \times 100 = \underline{\underline{20.8}}$

Miss rate = 1 − cache hit

** Miss penalty is nearly around 100 times than a hit

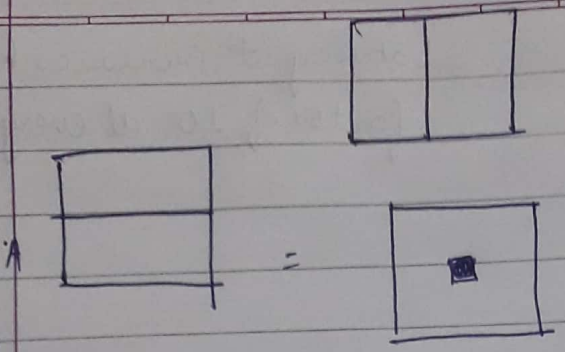
L1 - 32K

L2 - 256K

L3 - 20MB (cluster)

6MB (lab PC)

Page No.:
Date:



N
(10^5)

Matrix elements = 8 bytes

$O(N^2)$

Cache block size (C) = 64 bytes

Cache hit miss

$= \frac{N}{8} + N$ (column) $= \frac{9N}{8} = O\left(\frac{N^2 \times 9N}{8}\right) = O\left(\frac{9N^3}{8}\right) \approx O(N$

(Row)

Cache miss for serial

Block size $\sim B^2 = (B \times B)$

Cache can hold around 3 such blocks.

~~$3B^2 \leq C$~~ $3B^2 < C$

Cache miss for each block.

$\sim \frac{2N}{B} \times \frac{B^2}{8} = \frac{NB}{4}$

Number of blocks = $\left(\frac{N}{B}\right)^2$

Cache miss ~~for each blo~~ $= \frac{N^3}{4B} = O(N^3)$

Larger the block size quicker the processing.

Faster speedup = $\frac{9N^3}{8 \times N^3} \times 9B = 4.5B$

Page No.:
Date:
YOUVA

Lessons:

1) keep your working dataset very very small
2) keep your stride size small.

Problem:

T_p

n= p processor takes $(\log n)^2$ times ⇒ parallel
serial time is ⇒ $n \log n$ ⇒ serial

① speedup ② Efficiency

$$\text{Speed} = \frac{n \log n}{(\log n)^2} = \frac{n}{\log n}$$

Speedup ↑ problemsize ↑

$$\text{Efficiency} = \frac{n}{\log n} \times \frac{1}{n} = \frac{1}{\log n}$$

Efficiency ↓ problemsize ↑

Cost complexity ⇒ $n(\log n)^2$ ⇒ Not cost optimal by a factor of $\log n$

Let make $p << n$

Now the parallel time will be $\frac{p}{n}(\log n)^2$

$$\text{Speed up} = \frac{p}{\log n}$$

Speedup decreases as problemsize increases. Since it is not cost optimal