July 27, WEDNESDAY

# HIGH PERFORMANCE COMPUTING.

- Optimize run-time but also consider hardware.
- Asymptotic analysis.
- Not faith-based (No established laws) There are thumb-rules and guidelines but no absolute truths.
- There are parallel computing maps that are formed.
- Processors are called socket. Multi-core sockets.
- Number of transistors used decides performance
  HPC Cluster → # Nodes → Sockets → Core
- Based on the algorithms used, the code can be parallelized in different ways.
- Speed up vs. number of cores } Plots
  Speed up vs problem size

July 29, FRIDAY

FLOPS = freq. × $\frac{\text{no of op.}}{\text{clock cycle}}$ × no. of cores.

This is how performance is measured.

- Initially, the number of cores was not too important. The frequency could also not be increased much beyond a point. (Due to heat)
- Then, there were two new design philosophies:
  (i) Latency minimization (unit. sec.)
  (ii) Throughput maximization

- Latency: Time required to complete a task
  All CPUs are based on this
- Multi-core system. (4 - 20 cores)

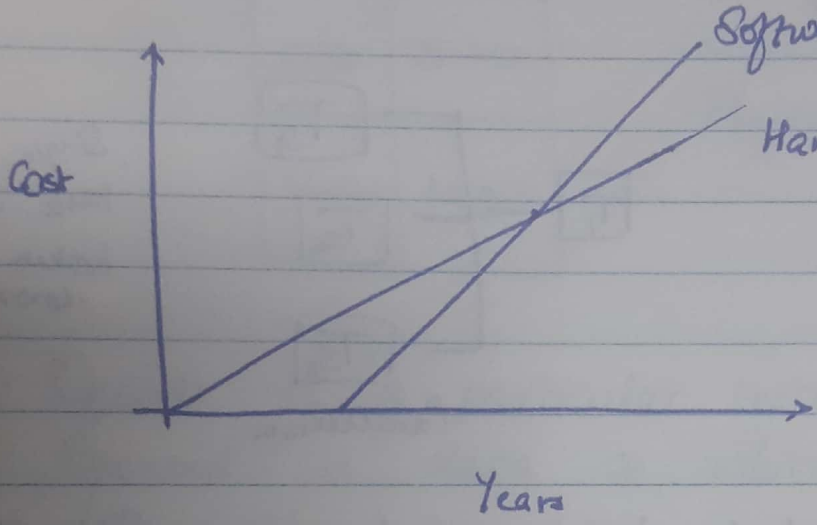
- Throughput: No. of operations per cycle.
  How much total work you can do in a given time. Does not matter how much time each operation takes as long as a lot of operations are performed
- Many-core (100 - 1000 cores)

- Practically, the design is a combination of both philosophies.

- For as many-core, you need a smarter/bigger control unit. Thus, in general, multi-core is used.
- GPU uses throughput max.

→ Factors affecting performance.

The code must be scalable and portable.




You cannot change software every time you change hardware.

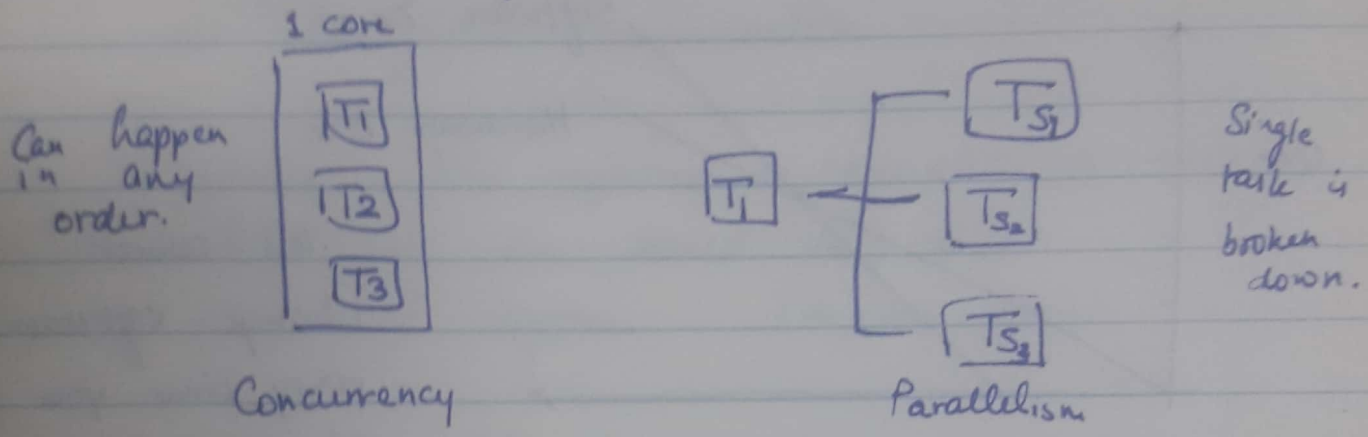
Portable: Can work on any hardware / platform
Scalable: Can work on a higher number of cores as well

- Memory access time decreases by shared memory.
- Bandwidth tells us about throughput. bytes/sec.
- The constraints is computer architecture. So, focus on parallelism

→ Multi-tasking (Concurrency) vs Parallelism

Time slicing and multi-scheduling leads to multi-tasking. Switching between

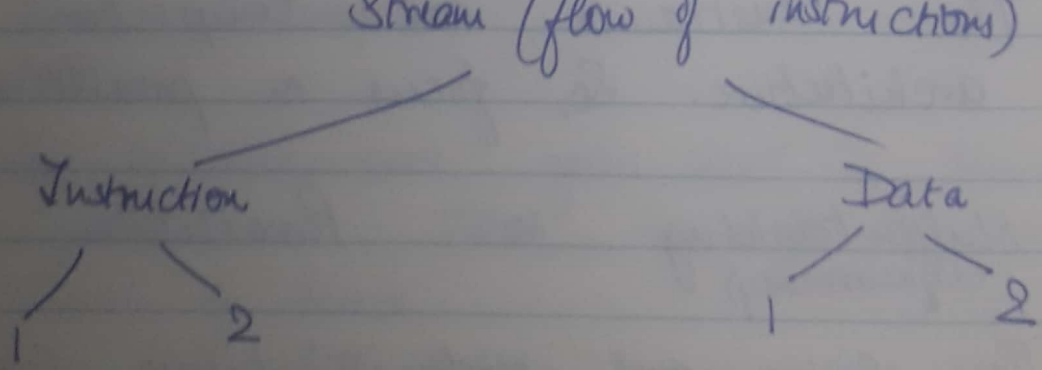
different process very fast isn't parallelism. You are only sharing resources. Both tasks run and neither is completed. In parallelism, both tasks run simultaneously.




Concurrency control is used for communication

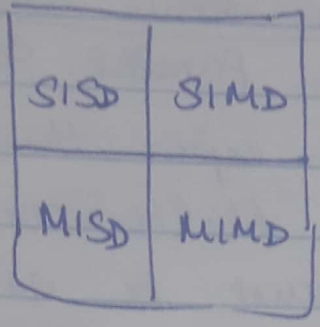
- For more performance increase power

August 2, Tuesday. There is instruction and data level parallelism.



SIS - Single Instruction Single
SIM - Single Instruction Multiple

These are serial instructions.




Flynn's Taxonomy.

- Opcode is a particular instruction.
- Operand is data on which we perform the instruction.
  Thus, there is always an opcode & operand.

- Pipelining: Some kind of assembly code
- Shared memory: Memory shared by diff. processors
- Distributed memory: Cannot access directly but through some network

Shared memory has very fast access.

- Granularity: $\frac{\text{Computation}}{\text{Communication}}$

The ratio is important because they happen at different time scales. Computation is much higher. Thus, we want a ratio < 1.
~~[Ideal. Not possible]~~

> 1 Coarse granularity
< 1 Fine granularity

- Observed Speedup: No unit. Ratio.

$$S = \frac{\text{Time on serial computer}}{\text{Time on parallel compute}}$$

Processors must be kept the same.
Single core vs. Multi core
Other hardware must be same to allow comparisons.

$$S = T_s / T_{s/p} = p \rightsquigarrow \text{no of processors}$$

Ideal. Doesn't usually happen.

If this relation is satisfied, linear speedup

There is also superlinear speedup ie higher than p.

If the problem is coarsely granular, you can increase efficiency by/to get (?) linear speedup

- Synchronization: Even after division, it might be required that some processor finish a task before some others

- Parallel Overhead:
  Launch threads, Kill threads.
  Explicit synchronization statements

- Massively Parallel: Huge number of cores

- Embarrassingly Parallel: High Granularity.
  These are the best algorithms.

- Scalability

eg. SIMD
Sum = 0 loop: 1→n Add 1.
Divide the array among different processors. Compute individually, store in private and then merge.
Efficient parallelization is not parallelization of the same algorithm but generating a new algorithm altogether.

- Why parallelism? Problems:
  - Communication among cores
  - Load balancing
  - Synchronization.
- Complexity does not necessarily remain the same. May / may not be better than serial

- Data parallelism & Task based.
  ↳ set of instructions

→ Amdahl's Law:

Maximum speedup is governed by

$$\text{speedup} = \frac{1}{1-P}$$

P = fraction that can be parallelised
S = serial part
S + P = 1

eg. If P = 0.95, speedup = 20.
P = 0 speedup = 1.

Theoretical maximum ~~=1~~ can be ∞.
P = 1.

Speedup depends on S.

$$\text{Speed up} = \frac{1}{P/N + S}$$

↳ no of processors

If N >> P,
P/N ≈ 0
∴ Speedup = 1/S

This is just theoretical. Does not account for parallel overheads

- Efficiency: Every time you increase the number of procession, your efficiency decreases.

Increasing the size of the problem decreases communication time (or no. of communication)

eg. size = n  3 procession

Ideally, if size = 2n, we would use 6 procession and thus 6 communication. But, using 3 procession limits communication to 3.

This increases granularity.

- We measure time of each part of the code and not the whole code. This is because entire code time might also include wait time. Thus, put some timing functions in the portions of code. This is called elapsed time. Resolution is the order of time that a function can entertain. Use appropriate functions (timer functions) with resolution for computing elapsed time.

August 3, WEDNESDAY. Linux commands: lscpu, /proc/cpuinfo for hardware details

Speed up = ~~T1~~

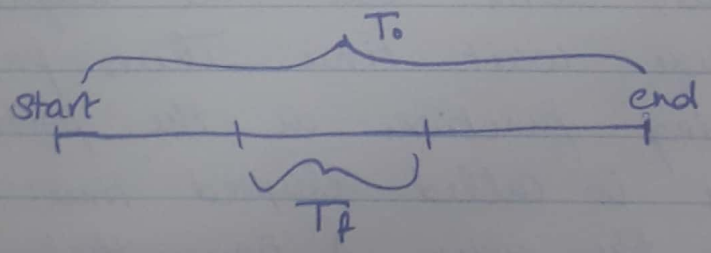
Cache memory for instructions differs from cache instruction memory for data

a, Change problem size } Look at the Speedup
b, Change core size } (Not necessarily linear)

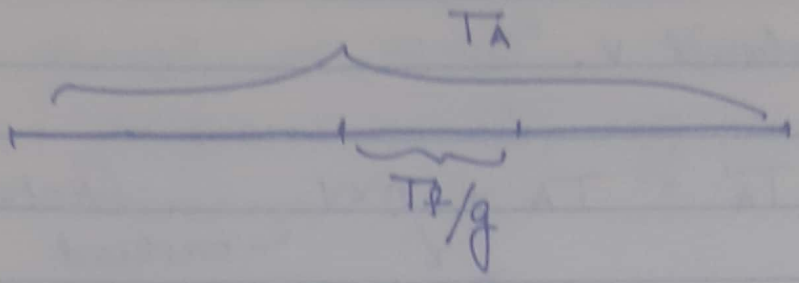
→ Optimizing the code.

To understand relation between code and hardware (time it takes for execution of code)

- Amdahl's Law.




T0: Time taken for serial code
Tf: Part of time taken by code that can be serialized parallelized
g: peak performance gain for accelerated portion




$S = T_0 / T_A$ (Speed up)

$f = T_f / T_0$ f: Fraction which can be accelerated.

$$T_A = (1-f)T_0 + \left|\frac{f}{g}\right| T_0$$

$$S = \frac{T_0}{T_A} = \frac{1}{(1-f) + \left|\frac{f}{g}\right|}$$

$$T_A = T_0 - T_f + \frac{T_f}{g}$$

$$T_A = T_0 \left(1 - \frac{T_f}{T_0}\right) + \left(\frac{T_f}{T_0}\right)\left(\frac{1}{g}\right) T_0$$

Theoretical

If we increase the problem size, the amount of overhead does not catch up with the increase in computation.

So, increasing problem size increases granularity

- Overhead v.

$$T_A' = T_A + n \times v \quad \text{(Actual)}$$

$v$ → overhead

Due to each of the section where we did the speed up.

- Measure overhead within the parallel computation

- Different Time functions have different precision (resolution)
- Else time function with much higher precision. (of μs)

$N = 10^7$

| Processor cores | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Speed up | 1 | ≈1.9 ≈ 2 | ×2.5 | 3.2 |

linear ~~Scale~~ speed up

$3 = \frac{3}{2}(1.9)$

$4 = 2(1.9) \approx 3.8$ but we get 3.2

Efficiency ↓ decrease.

↳ $N = 10^7$, → $N = 10^9$ Relatin Overhead decreases. So, efficiency relatively increases.

- Scalability → Strongly scalable
  → Weakly scalable

If we increase the number of cores and increase the problem size in such a way that the efficiency remains constant. Then, the problem is known as Scalable.

$T_{serial} = n$ sec.

$T_{parallel} = \left(\frac{n}{p} + 1\right)$ sec. → linear scale-up

problem size = n

no. of processes = p

$\frac{n}{p} >> 1$ Problem size $>> p$

$\frac{n}{p} >> 1$ So, $T_{parallel} = \frac{n}{p} + 1$

$\approx \frac{n}{p}$

$$E = \frac{\text{Speed up}}{\text{Procession}}$$

$$E = \frac{n}{\left(\frac{n}{p}+1\right)} \times \frac{1}{p} = \frac{n}{n+p}$$

If we increase procession k times, then we will have to increase problem size 'x' times such that efficiency remains same

$$E = \frac{xn}{xn + kp}$$

$x = k$ for efficiency to remain same.

If we are not able to find an 'x' such that efficiency remains same then it is 'weakly scalable'

- Profiling: Information about program's behaviour

Run-time → "hot-spots" → performance bottle ~~neck~~ neck

are areas which take more time and we should focus on these.

↓

Optimization.

- Function profiling:
  If short functions, then we look at time taken by the function.

In operation - how we are accessing data. Essential to look at this.

eg. gprof from GNU.

i. Flat function profile
ii. Call graph profile (butterfly graph)

gprof: Compile with ~~f95 -pa~~
f95 -pg file.f

i. Flat profiling:

./a.out then we get gmon.out

The information profiling is sorted by 3rd column
The 1st column shows percentage contribution of each method.

August 5, FRIDAY

Problem size = n ,

$T_{serial} = n$

$T_{parallel} = \frac{n}{p} + 1$

If we increase the processors, then problem size has to be increased.

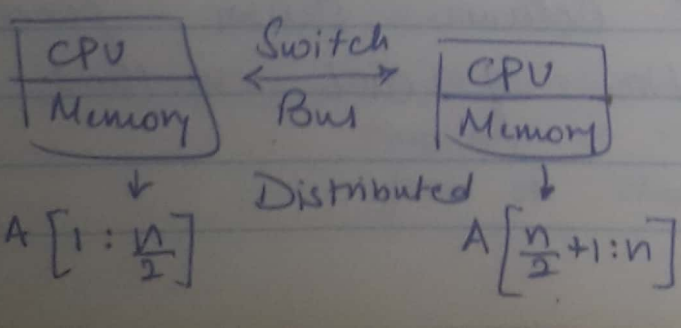
- Strongly scalable: If we increase number of processors without increasing the problem size and efficiency remains same
- Weakly scalable: If we increase number of processors and we have to increase the problem size so that efficiency remains same.

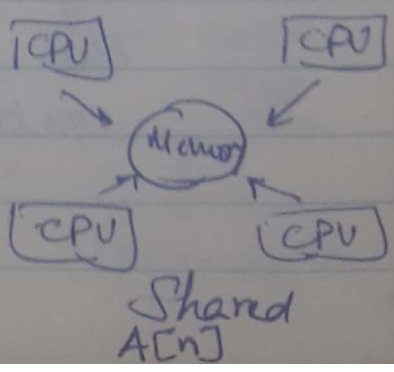
→ Shared Memory Parallelism.

Open MP - Multi processing.

* pragma omp { ... }

This can be used only when all cores have shared memory.



In order to access the other memory, we require to communicate with it using switch/bus (Explicitely pass messages)

we have to ensure that there is no collision (synchronous access) to the same memory.

- Fork: brings concurrency.
  Creates child threads which run concurrently with parent
- Join: Removes concurrency.
  Child joins with parent process. Parent waits for child to exit.

C:
```
# pragma omp parallel
{
  // code to be executed by each thread.
}
```
(Each thread is launched on different cores)

- Specify Threads: ~~Inside the code,~~
  In code, Setenv OMP_NUM_THREADS = 4
  In bash, export OMP_NUM_THREADS = 4.

Compiler Directive

- Control construct (Launching threads)
- Data Scoping (Race conditioning, Shared access)
- Synchronization

August 9. TUESDAY.

- Structured block is a part of code that have one entry point and one exit point.
  No branching.
  These are easiest to parallelize.
  With branching via "if" statement, cannot determine exact flow.
- Take a serial code and make minimum changes. This should get maximum change in speed-up.
- Trying to discretize a continuous function leads to computational inaccuracy.

eg Integration.

```
h=(b-a)/n;
approx = (f(a) + f(b))/2.0;
for(i=1; i<=n-1; i++){
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

[To compile: gcc -fopenmp hello.c]

To parallelize, we can assign some trapezoids to some cores, thus dividing the work. Each core individually computes a private result and all these are added at the end. to get the final result.

global-result += my-result.

Mutual exclusion. Race Conditions)
Problems of concurrency will arise when multiple threads try to update it. Thus, add mutex() equivalent
critical directive ← #pragma omp critical.

- If you make my-result() a function, we are actually serializing it because that function will run one at a time.
- If my-result is an array, when trying to access [0][i] simultaneously, it tries to access cache lines which creates a problem (TBD)?

* Default scope of variables declared before a parallel block is shared.

→ Reduction:

Reduce to a single reduced variable.

? Similar to calling piece of code critical.

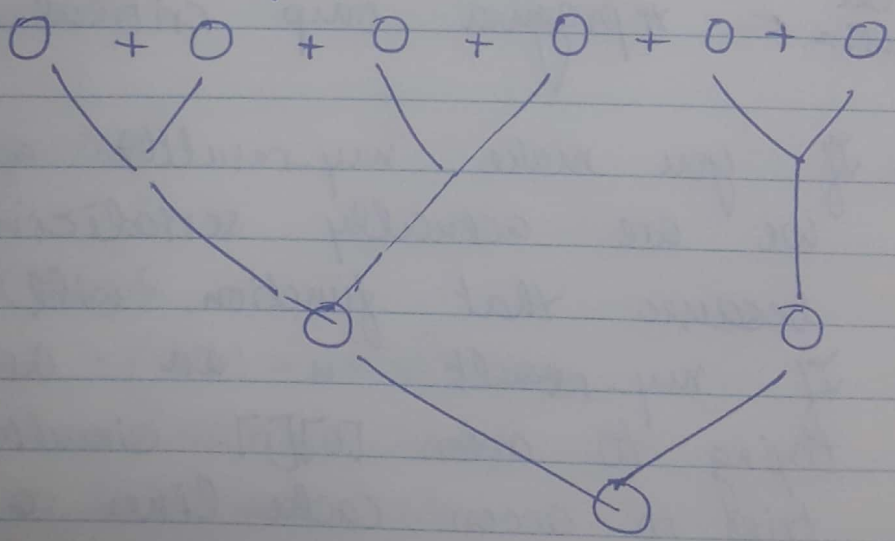
- Floating point in binary is not associative.

reduce(<operator>: <variable_list>)

- Exactly like critical but the compiler does it for you.

Binary operators.

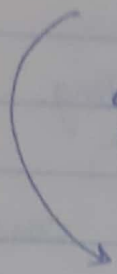
- Reduction algorithms.




Thus, still some threads can work in parallel while updating.

More efficient, requires less changes, than 'critical'

- Dependency algorithms: Programs where the next step cannot be computed before the previous one.
  eg. Fibonacci.
  You must remove all loop dependencies.



This is loop dependency.
Data dependencies may be fine.

#pragma omp for

This specifies sections where the code has to work. Improves efficiency.

August 10, WEDNESDAY.

Intel profiling ← Commercial profiling.
Use times instead of gprof.

→ Optimum Techniques:

eg. if do( )
    if(a > 2)
A)  b exit
    endif

do ( )
If (a>2)
B) endif

Use A) because break ~~ends~~ once use of loop is over.

eg. `A = B**2.0` → Exponent is actually calculated by $x^y = e^{y \ln x}$

`A = B*B` ✓ More primitive so faster

Thus, more demanding.

eg ~~`for i=1,180`~~

`A=0`

`i=1,180`

`A=A+h` `h=1`

`cos(A)`

To compute cos() function faster, store the values in a look up table. Thus, we can have a ~~the value~~ table for all values in the cache and access is direct.

- Cache exploits locality → {Temporal, Spatial}

Look-up tables are generally stored in 1st / 2nd level of cache.

CHECK: How many clock cycles does division require vs multiplication? 4

eg. Prediction

- Branching is costlier and takes more time ~~for predictions~~.

eg.
```
do (     )
   do (     )
      if (     )
      else if (     )   } Branch
      else
```

- Each processors has its own cache. Thus, when you optimize for one processor, optimize for all processors.
  This leads to superlinearity.
  (Speed-up becomes more than theoretical as theory does not account for multiple cache)
- Different languages use different defaults in terms of storage.
  C is row-major
  Fortran is column-major.
  Thus, this would change the spatial locality in memory.

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & & \\ M_{30} & & \end{bmatrix}$$

Row: $M_{00}\ M_{01}\ M_{02} \rightarrow M_{10}\ M_{11}\ M_{12}$

Column: $M_{00}\ M_{10}\ M_{20}\ M_{30} \rightarrow M_{01}\ M_{11}\ M_{21}\ M_{31}$

→ **Scheduling:**
How many jobs are assigned to each thread? How this is broken apart.

- **Types:**
  i. Static: Simply equally divide. If not exactly divisible, give extras from the beginning threads.

This would be load imbalance.

Scheduling done at compile time

- May be possible that slower thread gets more.
- We are not aware of how the hardware exactly works. Thus, some threads might work faster than others if the hardware of that core is better.

(i) Static - 3 : Specify chunk sizes. ?
~~Extra~~ That much allocated to each

(ii) Dynamic: Based on ~~abilities~~ availabilities of that ~~the~~ core, jobs are assigned to the threads.

Not work at run-time

(iii) Guided: Looks at the works and divides and assigns. Then, remaining work is ~~assign~~ divided again and assigned. Keep dividing.

→ Work Sharing:

```
x = 50
#pragma omp for private(x)
{       i = 1, 20
        x = i
} print x.
```

Specifies that work on x which is in stack of thread.

Ans = 50

lastprivate() → Stores the value of the last division.
This doesn't mean that the last thread is supposed to be the last thread computed. But, value stored is the last one.
Ans = 20.

August 16, TUESDAY. "Computational science is all about linear algebra"
Three combinations: Vector - Vector
Vector - Matrix
Matrix - Matrix

→ Vector Dot Product:
Multiplication of two vectors followed by summation

Private variables are stored in individual stacks.
Shared are in heap.
This is the same as local and global.

#paragma omp parallel for default(shared) private(i)

Is private() necessary?
Find difference between global and shared variables.

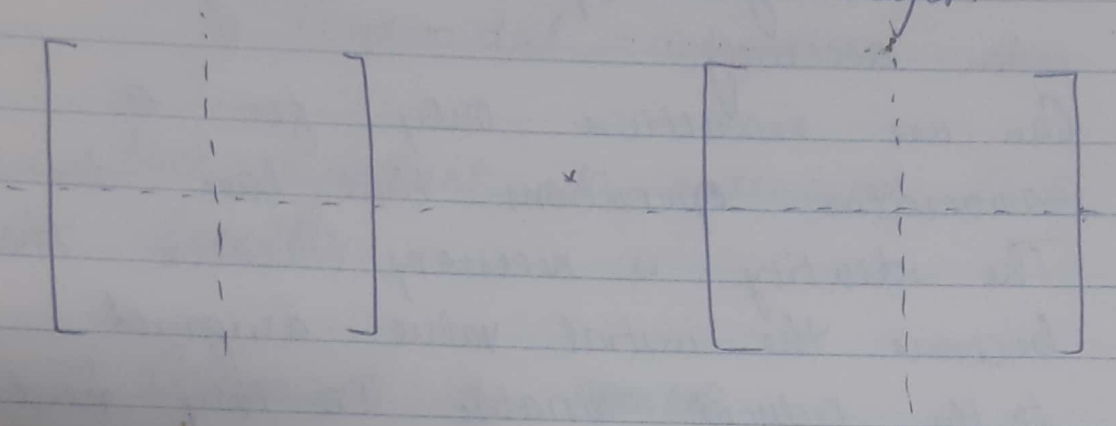
collapse(2) For nested loops.
Makes single loop of NxM and parallelizes.

of second
No. of columns must be equal to number of rows in first

Sequential code:
Each statement does 2 memory accesses and 2 operations All of this affects the performance.
$O(n^3)$

- While parallelizing, the main issue is data access. This is what will adversely affect the speedup.
  Try and see the pattern of data access and if any part is re-usable. Once read, see if the value can be reused?
- Instead of reading whole rows and columns, divide the data into blocks and then compute in partitions.
  This is called Block Matrix Multiplication.
  This reduces data access time.
- Cache is used to reduce data access lag.
- When an or matrix goes into cache, locality is also very important. Thus, when you partition the matrix into blocks, the spatial locality of each block greatly increases. Thus improving speedup

Also have to think about which of the three loops you will parallelize. For this, look at how C ~~or~~ traverses arranges matrices ie ~~Column~~ row major.



- Coalescing also takes place. This changes depending on the language you use.
  eg C is row major. So, if you have A[0][0], A[0][1], A[1][0], A[0][0] and A[0][1] will coalesce.
  Coalescing means combining adjacent blocks.
  ? Thus, ~~how~~ what blocks are made affects the time taken for computation.

Assgn 2: $\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$

sum = 0 → Shared.

This needs to be mutually exclusive.
Three ways to do this:
Critical
Reduction
Atomic (?)

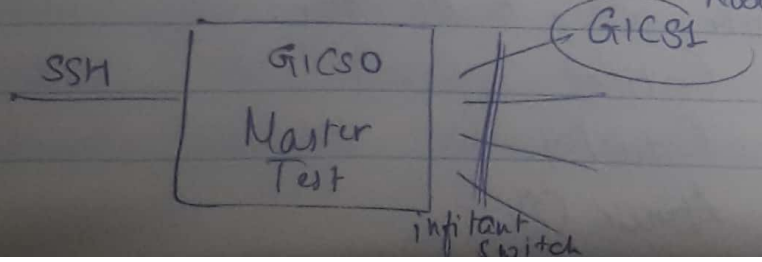
reduction(+; sum)
When using reduction, sum becomes a private variable. It is private when being computed but global when accessing.
Can use reduction only for associative operations that have identity. The identity is necessary because that becomes the initial value assigned to the reduced variable. Do they need to be commutative? No

```
sum=0;                    reduction(+; sum);
for i=0,N
{
        sum += (-1)^k / (2k+1)
}
```

How do you ~~use~~ use the k? It is dependent on previous but won't be valid during threads.
Can use i%2.

August 19, FRIDAY.

ssh <ID>@10.100.71.130.



Classify problems into various categories and approach is different for each.

(i) Divide and Conquer
eg Trapezoidal integration

- Most important to understand Cache and locality.

→ Components of OpenMP
(i) Directives → work-sharing functionality.
(ii) Runtime libraries → no. of threads, thread ID, nested parallelism, timers.
(iii) Environment variables → setting no. of threads, scheduling

OMP_DYNAMIC → Breaks the code into chunk sizes.
Scheduling → Chunk size
→ static/dynamic

There is always an optimum chunk size.

- Implicit barrier while existing a parallel section is cannot exit if all threads have been computed.
If dont not, no-wait()

*pragma omp master
Only the master thread is allowed to access the share variable.

→ Embarrassingly Parallel Computations:

Can be divided into completely independent parts. The threads do not need to communicate
eg Matrix addition
Random number generator.
Image processing (Geometric transformation of each pixel)
Most important is how you divide.
At times, may get worse results than Serial if not divided correctly.

Like a 2D-array. 2 loops required to traverse. Each pixel has 3 data points (R, G, B). This is converted to a black and white picture where each pixel only carries the intensity
Data access has to be considered.

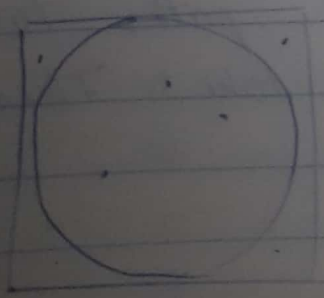
$$x' = (x-c_x)\cos\theta + (y-c_y)\sin\theta + c_x$$
$$y' = -(y-c_y)\sin\theta + (y-c_y)\cos\theta + c_y$$

Colour scaling is more efficient than transformation. Why?
? The location of the write pixel changes and so data access takes more time.

When we run the program, data is shifted from main memory to RAM.
Two types of registers:
(i) General purpose
Data storage. Operands
Can it have instructions?
(ii) Special purpose
PC, stack pointer, intruction register
current instruction.

What happens during ./a.out? What goes into RAM?
exec file?

eg. Monte Carlo method for $\pi$



Generating coordinates randomly.
Finally, look at probability of the point lying inside the circle.

- Uniformly distributed: Equally dispersed over the number-line. No bias.
- Statistically independent: Future value is not dependent on previous values. This makes it much easier to parallelize.
- Can use histograms to check uniform distribution. Can also check average of all. How to use random number generator for parallel programs? Does it matter? Go to same random?

August 23, TUESDAY. Memory access is very important. Have to look at latency (time taken for one access)

- If you request 0 bytes, bandwidth (byte/sec) is 0. However, latency is not zero. Some fixed minimum time will always be used.
- Memory heirarchy.
  The actual distance from the CPU is looked at. L1 closer than L2. This is what determines the latency.
- Main memory: DRAM (Dynamic RAM)
- There are many different paths to reach that the program can take. This decides performance.

- General purpose  Special purpose registers
  ↘ A program can access only them.
  Two important instructions: STORE, LOAD.
- To probe the memory hierarchy and see when saturation happens (proportional to problem size) (Peak performance which in turn is measured through speedup)

→ Microbenchmarking:

Difference between processors: instruction sets
Basic form of parallelism: (i) System level parallelism
(ii) Instruction level parallelism (ILP): pipeline
Breaking down the work (instructions) in steps which can be done in constant time.

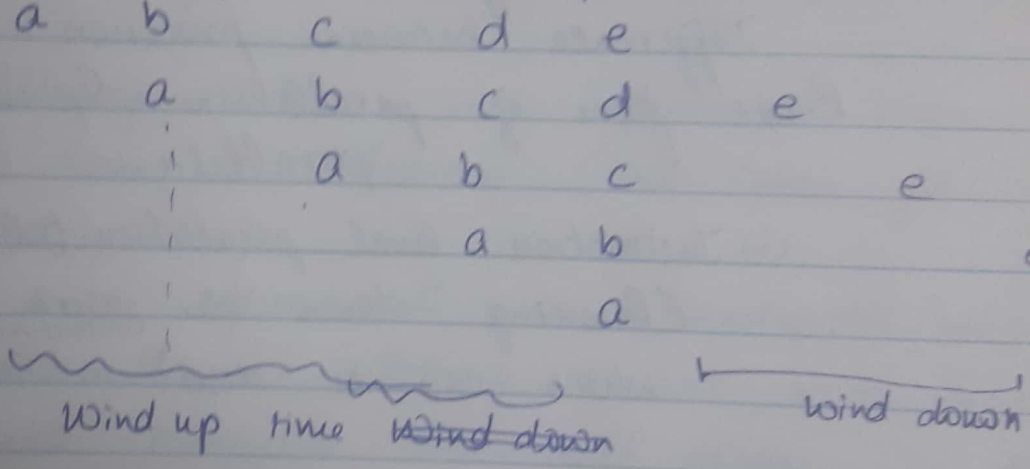
Depth of pipeline { 1, 2, 3, 4, 5 } → Stages of pipeline

Simplest: Fetch Decode Execute

| Clock | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | a | - | - | - | - |
| 2 | b | a | - | - | - |
| 3 | c | b | a | - | - |
| 4 | d | c | b | a | - |
| 5 | e | d | c | b | a |
| 6 | - | e | d | c | b → a |

- Upper triangle of pipeline matrix is called Wind down. This is the part where instructions are leaving. Overhead.
- First half is wind up, once no new instructions enter. This is also Overhead.

- Longer pipelines implies greater overhead.
- problem size → n=1000 loops better than n=100 (nested loop ↑ wind up ↑↑)




Wind up is the time taken for an instruction to be executed.
Wind down is the time the last instruction takes to be executed.

Overhead = Wind up + Wind down

- After wind up, we will obtain one instruction per clock cycle. All stages of the pipeline get filled.

Wind-up and Wind-down take much more time if the pipeline has more stages. During parallelization, they can be considered as overhead.

+, −, * → Diff. operators. Look at time of execution.

$T_{exec} = E - S$

$\frac{T_{exec}}{N}$ = Performance of diff operators

August 24, WEDNESDAY

Hardware:

Size of cache memory, bandwidth for cache access, bus speed, processor architecture...

- Difference between low-end (computer) and high-end (GISC) machines.
- Cache size is larger for high-end machines (Xeon) making parallel code performance much higher.
- Lifetime of a thread stack is the lifetime of the thread. (Variables are lost after)

- Stacks are faster than heaps. But stacks have limited size (They can also be extended?)

$i = 1, N \qquad C[i] = A[i] + B[i]$

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 |
| 2 | | 1 | 2 | 3 | 4 |
| 3 | | | 1 | 2 | 3 |
| 4 | | | | 1 | 2 |
| 5 | | | | | 1 |

Latency

Wind Up = 4

Max. throughput = 1 instruction/cycle.

- We can get maximum throughput after wind-up
- For n-depth pipeline, we obtain max throughput after (n-1) cycles.
- Pipeline for different functions is different. (? What happens to wind-up & down)

eg. $n = 5$
Depth = n
Instruction = N

ILP, cache, superscalar architecture = Throughput more than 1

$$\text{Speed up} = \frac{nN}{N+(n-1)}$$

N instructions, each takes n cycles, Serial so no pipeline $nN$

Pipeline is the parallelization here. ILP. $T_{parallel} = N+(n-1)$

$(n-1)$ ↓ Wind-up cycles (overhead)

For complicated instructions, pipelines are much longer so latency is much higher.

$$\text{Speed up} = \frac{n}{1 + \frac{(n-1)}{N}} \qquad (n-1) << N$$

In this scenario, $(n-1) << N$ speedup $= n$

(Speed up $\propto$ Depth of pipeline)

$$\frac{n-1}{N} \to 0$$

$$\text{Speed-up} = \frac{nN}{N+(n-1)} = \frac{(n-1)}{(n-1)} + \frac{\left(\frac{1+N}{(n-1)}\right)^{\to 0}}{\frac{N}{(n-1)} + 1}$$

$N << n$

$\therefore S \to 1.$

For example of 'N' addition,

Throughput = No. of operations/cycle

$$= \frac{N \text{ operations}}{N + (n-1)}$$

(pipeline depth)

For parallel code,

No. of cycles = $N + (n-1)$

No. of operations = $N$ additions.

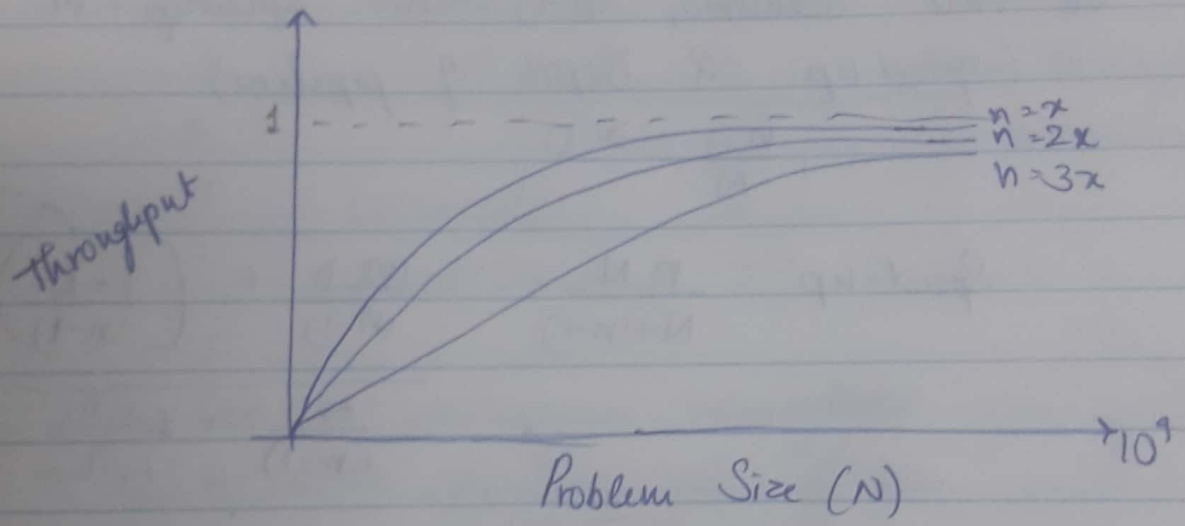
Speed-up = $n \times$ Throughput.

**Speed-up = Depth x Throughput**

Throughput $= \frac{N}{N+(n-1)} = \frac{1}{1+\frac{(n-1)}{N}}$ → no. operations

Addition → Multiplication → Division.

·Depth(n) of the pipeline changes.




Smaller depth will saturate to maximum throughput faster than operations with larger depth. (As has lower overhead)
So, we should try and use simplified operations with smaller depth.

→ Loop Dependency.

eg. $A[i] = A[i-n]$

$A[1] = A[0]$
$A[2] = A[1]$

MOV A[i-1], A[i-2]
MOV A[i], A[i-1]
↳ Because second instruction, problem in pipeline. Have to wait for first inst to be complete before executing

MOV A[i], A[i+1]
MOV A[i+1], A[i+2] → Since first, doesn't matter. Don't wait. Can directly execute without problem. A[i+1] is updated only after.

If n = -1 → Real dependency
n = +1 → Pseudo dependency.

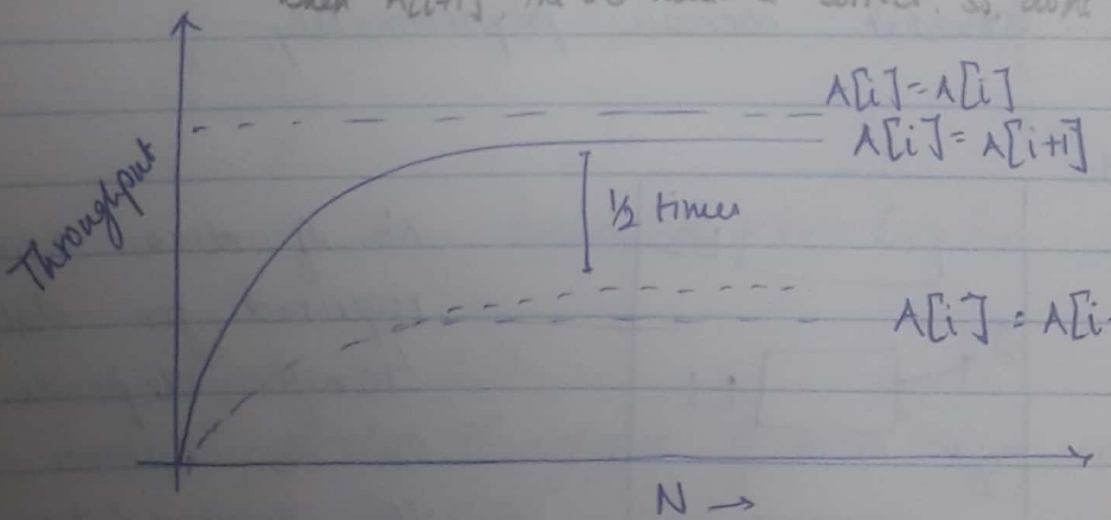
A[i] = A[i-1] Until A[i-1] is not completed, A[i] won't be either.

A[i] = A[i+1] → Faster. Don't need to flush
A[i+1] is already present which can be used to directly update A[i]
Value is available so update is done without waiting.

Mem read | Fetch | Decode | Execute | Mem write

A[i] = A[i-1]
A[i-1] = A[i-2]

Talking about pipeline & flush.
When A[i] = A[i-1], it is possible that old value of A[i-1] is stored in A[i] execution stage. This needs to flush pipeline to get updated value. Then execute. So, more time
When A[i+1], the old value is correct. So, don't flush.



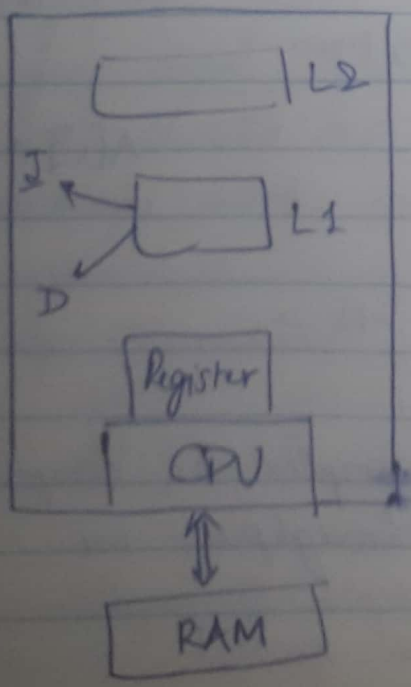

CISC is always complex as compared to RISC
But for larger throughput, we use CISC ? & Lesser depth of pipeline required
So, wind-up and flush quicker

Super Speedup architecture → more than one instruction per cycle.

Not able to give data too fast enough but processor speed keeps on increasing
DRAM GAP.

- Von-Neumann Bottleneck
  Limitation on throughput caused by the standard personal computer architecture caused by the standard personal computer architecture, like slower memory access
  Faster a processer is, more time it has to spend idle as the memory access time wouldn't decrease proportionately.

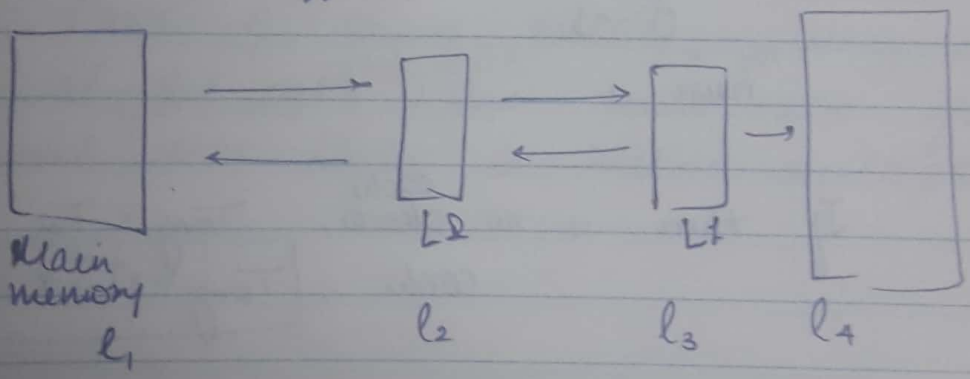
August 26, FRIDAY.




No. of clock cycles required for data transfer depends on the distance of the memory from the CPU.

Suppose bus has a clock rate of 600MHz BW) width = 32 bits ?

- Latency : Time for a single operation
  Memory latency as well.
  Time delay to transfer 0 bytes of data (which depends on different devices (memory units))
- Delay before actual data is put on line. Also, different distances of RAM, from CPU result into different latency cache




latency: $l_1 > l_2 > l_3 > l_4$

1) Pipeline : ILP
2) Superscalar: Throughput of more than one
3) RISC to CISC
4) Out of order execution - GPU (Compiler takes care)
5) MIMD

- Performance of CPU is increasing at a much higher rate than memory access rate
  → Processor - Memory Gap
- Temporal locality - Reuse of same cach data (over time)
  eg. repeated access a[i]
  Temporal → spatial locality.

Memory → $T_m$

Cache → $T_c = \frac{T_m}{\alpha r}$

If reuse ratio = 1/2
Cache hit $T_m$
Cache takes double the time than fetching required.

Reuse ratio → r

0 → 1

No reuse ← → Complete reuse (Theoretical)

If there is no cache, $T_{avg} = T_m$

cache, $T_{avg} = rT_c + (1-r)T_m$

- Cache line: Whenever we are accessing from main memory, the cache also copies the adjacent locations to the value we are searching
  Typically. 64-128 bytes.

For matrices, rows in C:

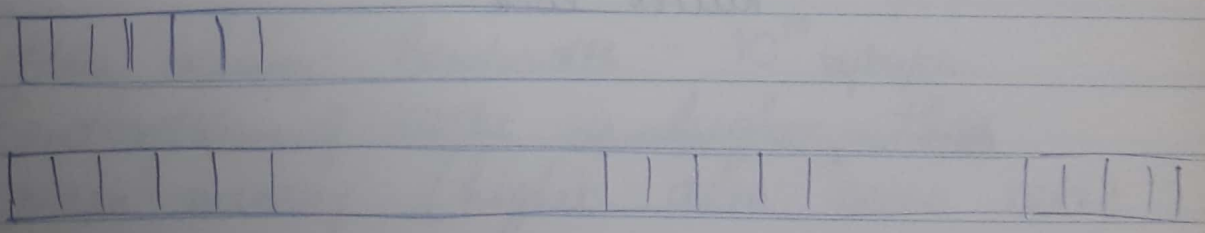
eg. Data = 128 bytes  Bandwidth = 50 GB/s
Latency = 50ns

$= \frac{128}{50} \times 10^{-9} \approx 2.4$ ns

Here, latency > Tdelay due to bandwidth,
Thus, it becomes a more critical issue.

Cache length = 16 bytes
Hit ratio = 0.5
→ We can find the data only half the time.
So, actual utility = $16 \times \frac{1}{2}$ = 8 bytes
(50% of spatial locality is being used)

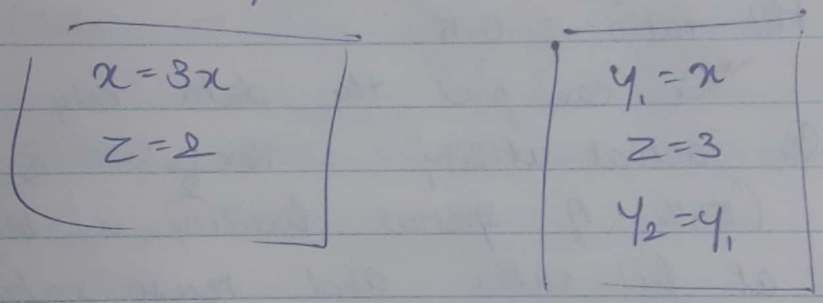
- Look at hit ratio and reuse ratio for different size blocks in matrix multiplication



Prefetch: To predict which data/instruction would be more useful.
Done by compiler. It will bring that even before requiring data/instruction.
Cache bandwidth, cache speed, cache throughput

- Why use smaller?
  Even if locality is not required in the program, we might end up blocking the whole bus.

- Cache Coherency:
  Changes in each cache should be reflected in memory.




In order to avoid race conditions, mutex lock.

barrier, atomic no wait

master single collapse

sections first private thread private

August 30, TUESDAY

Data Access: Change the datapaths based on which we can change performance.

→ Performance Modelling:

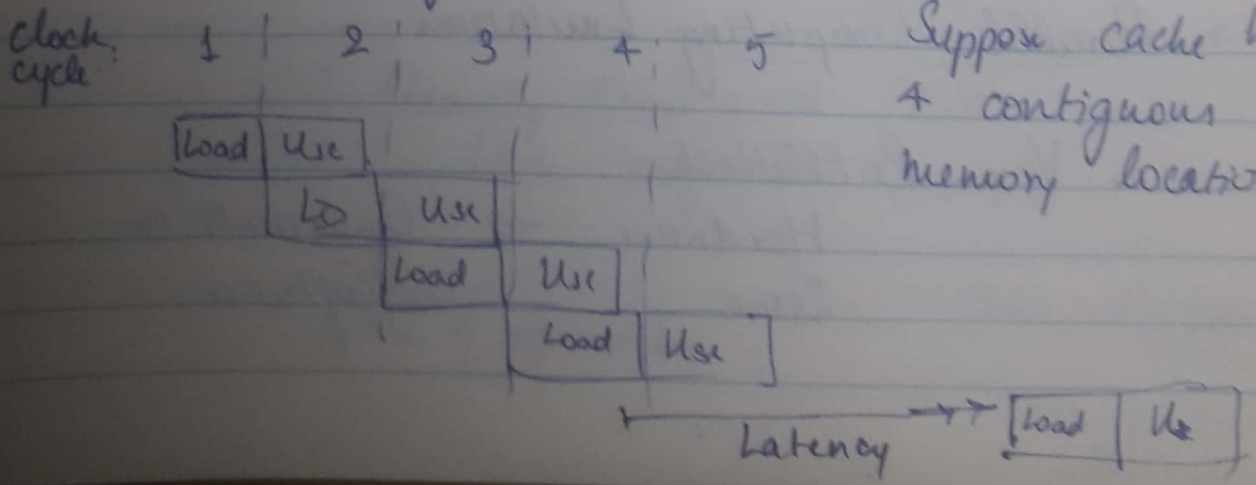
(i) Latency Delay
process, as well as memory

(ii) Bandwidth of path

- How do we reduce latency?
(i) Instruction level parallelism
(ii) CISC → RISC (smaller instructions so quicker)
(No. of operations reduced)

- Main memory: Bandwidth = $10^{10}$ bytes/sec.
- Bandwidth of cache is higher than main memory (higher data access rate)

→ Overpowering Latency:




Suppose cache has 4 contiguous memory locations

- If there is a cache miss, the data will be obtained after a latency (due to access from memory)
- In order to reduce latency, compiler prefetches the instruction. (Compiler predicts the possibility of usage of the cache miss element and thus starts prefetching and so in turn reduces latency)

→ Rule of 72.

Growth rate = x%

$\frac{72}{x}$ years will be required to double performance

DRAM : $\frac{72}{9}$ = 8 years 9%/year

CPU processor: $\frac{72}{60}$ ≈ 1.2 years 60%/year.

Thus, Processor-Memory Gap.

→ Parallel Computing: Limitations

i) Von BottleNeck:

Hardware → Constraints in Issue

Software

(ii) Moore's law:

More transistors → More Speed
↓
More power
↓
More heat → Less Speed
(CPU starts giving less performance)

(iii) Microbenchmarking:

a. Data Access
b. Processor's capability
c. Understanding hardware
d. Programming capability

→ Balance Analysis:

(i) Machine Balance
(ii) Code Balance

(i) Machine Balance: It is intrinsic.

= Memory Bandwidth (B/s)
Peak performance (FLOPs/s)

= Bytes
No. of operations

FLOP = No. of operations / cycle
× No. of cycles / second

- Depends on which memory you access (Bandwidth is different)

$$\text{Balance of RAM} = \frac{10^{10}\ B/s}{3.3\ GHz \times 4 \times 1}$$

(4 → No. of cores; 3.3 GHz: No. of cycles / second; 4: No. of operations / cycle)

Machine balance is going to decrease in the future as DRAM gap increases.

(ii) ~~Code Bo~~ eg. $A + B \times C$

Data Traffic = 3

No. of operations = 2

(ii) Code Balance:

$$= \frac{\text{Data Traffic (B/s)}}{\text{Floating point operations}}$$

$$\frac{1}{\text{Code Balance}} = \text{Computational Intensity of the code.}$$

We always want to do more floating point operations with the data. Hence, for better performance, code balance should decrease.

- Machine and code balance have the same units
- Data traffic: Performance limiting data path (hardware dependent)

  Data ← Load/Store

  (light speed !)

* Peak performance of a code is the expected maximum fraction of peak performance of a code with Balance code $B_c$ and Machine Balance $B_m$

$$\text{Light speed} = \min\left(1, \frac{B_m}{B_c}\right)$$

* Expected maximum fraction of peak performance of code = 1.

$B_m$ is constant.

If $B_c \downarrow$, $\frac{B_m}{B_c} \uparrow$

If the DRAM-gap increases, light-speed decreases.

i) Data access
ii) Processor
iii) Performance of Algo.
iv) Pipeline
v) Peak until latency hiding

- Our codes generally underutilize hardware. ← Serial code.

Parallel tries to overcome this.

GPU → Dynamic (Reduces power when not used)
GPU → Static (Uses static amount of power)

September 6, 9 Parallel Vector Triad benchmark
TUESDAY

```
do j=1, NITER
   do i=1, N
      A(i) = B(i) + C(i) * D(i);
   enddo
   if if ( ) then
      ( )
enddo
```

Code Balance = 4 (load/store data access) / 2 (No. of operations)

(assignment is not an operation because it is a 'store' operation. Under data access)

Bc = 2.

We want Bc to be as low as possible.

[Loop fusion, jamming, unloading

eg.
```
for i=1, N {
   A[i] = B[i] + C[i]
}
```

```
for i=1,N {
    D[i] = B[i] + E[i]
}
```

Increase speedup by loop fusion

Lightspeed $l = \left(1, \frac{B_m}{B_c}\right)$

$\frac{B_m}{B_c}$ shows what percentage of $B_m$ we are able to utilize.

"peak" performance.

Your machine can be more efficient, but you are only able to use $B_c$ portion of it.

[Look at this during the assignment/project]

Only way to improve after a point is by improving data access.

eg. $A_{present}[N]$ $A_{past}[N]$

After some function, say we need two array information to be stored.

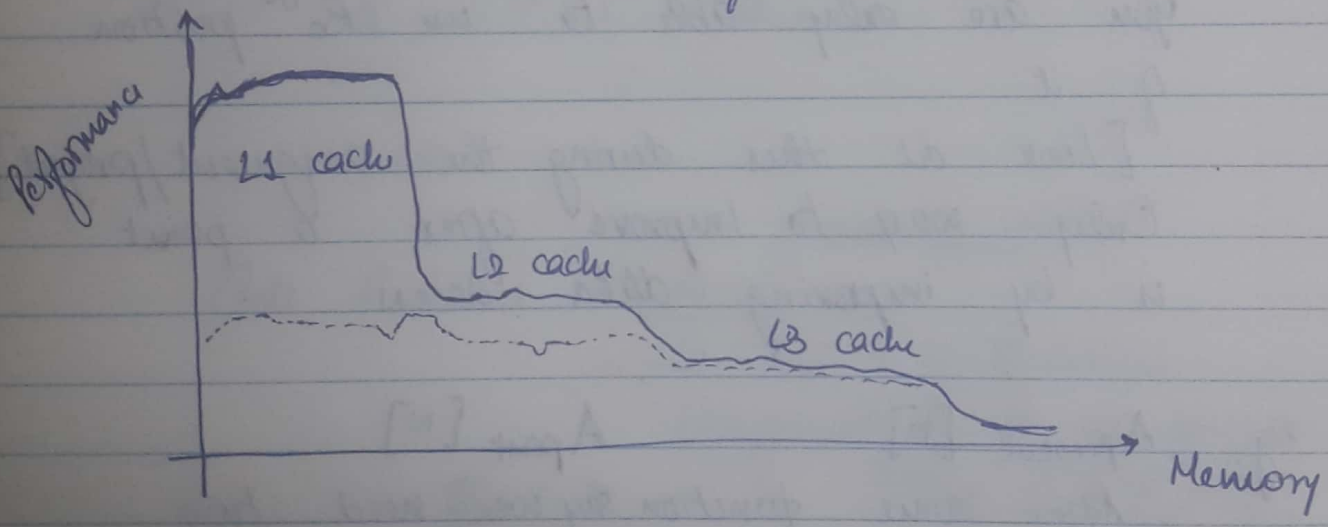
Rather than making them two separate arrays, it is much better to define a single two-dimensional array.

This has a greater chance of cache hits.

$A_{pp}[N, 2]$

- In the parallel vector benchmark, the outer loop has been added just to increase the code size so that looking at performance is easier. The bottom 'if' is necessary because without that, the compiler is smart enough to ignore the outer loop.

- Vector processing: The processor takes chunks of data instead of single points. This makes a huge difference




There is a small dip in the initial part. This is due to latency and bandwidth. Bandwidth becomes a problem only for larger problems. However, the latency exists even for smaller problems. Always have to wait.

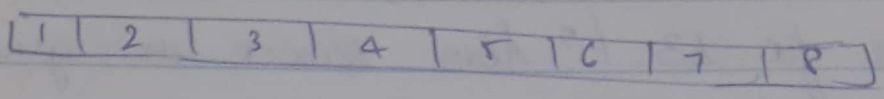
- There is a steep drop between the cache processings. This is because L2 has to interact with both RAM and L1 and so on for L3. (? Also slower in general)
- The initial drop is due to the wind up time in pipelines. The performance will increase only once the size of the problem is greater than the depth of the pipeline. Prefetching, etc are problems.

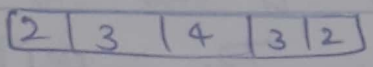
Ideally, we would have thought that the graph should have started off at a higher performance and then ~~decrease~~ saturate. because the whole (? data) program would fit in the cache and so less misses.
However pipeline plays a more important role.

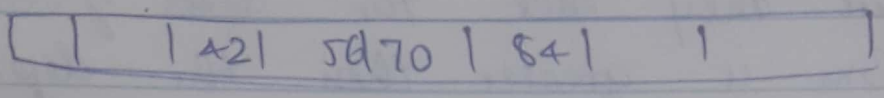
* Look at performance.
  Bm/Bc, latency, etc.

Some algorithms don't have a scope of optimizing beyond a point. So, you may have to change the algo as well.

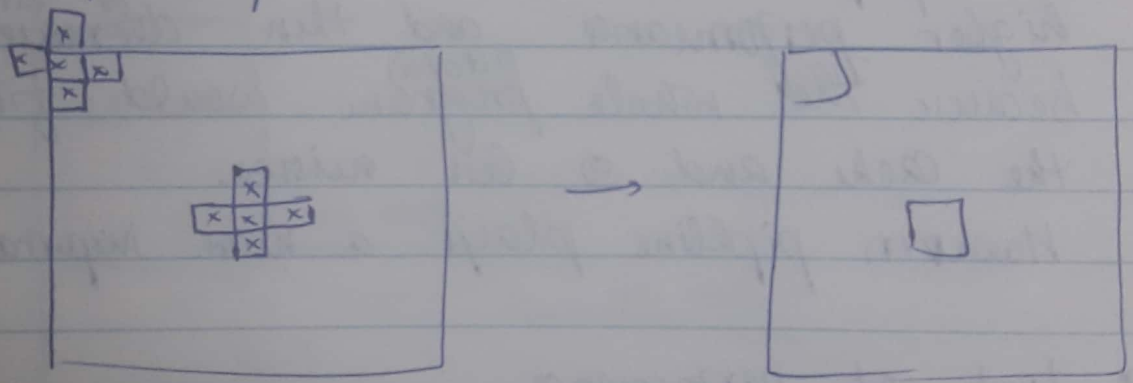
eg. Image processing




Masks and other things have odd sizes in general. This is because it has symmetry




The mask acts in the form of a convolution & Multiply each of the mask to each of the input and add. (Five terms at a time)

If you have to exceed the input, padding based on how long the mask is.
(ghost elements. Generally assumed to be 0)

Mask length = $(2N+1)^2$

- The filter/mask is constant. It won't be changed every time. It is already loaded into cache. Don't have to refetch. Thus, when you increase the size of the filter, you are increasing the number of operations per cycle without increasing data access.
  Increasing the image size changes data access time.

- Arrays are generally not stored in registers. They are kept in cache.
  A[i] + temp ~ s in register. Faster
  A[i] + B[i] No one in ~~a~~ register/cache.
  → Can be considered to be one load/store as access time for temp is very low in comparison (Low latency)

- Latency is a more important factor as compared to bandwidth.

3D convolution - used in MRI scans, etc.

September 7, WEDNESDAY

# Performance Modeling

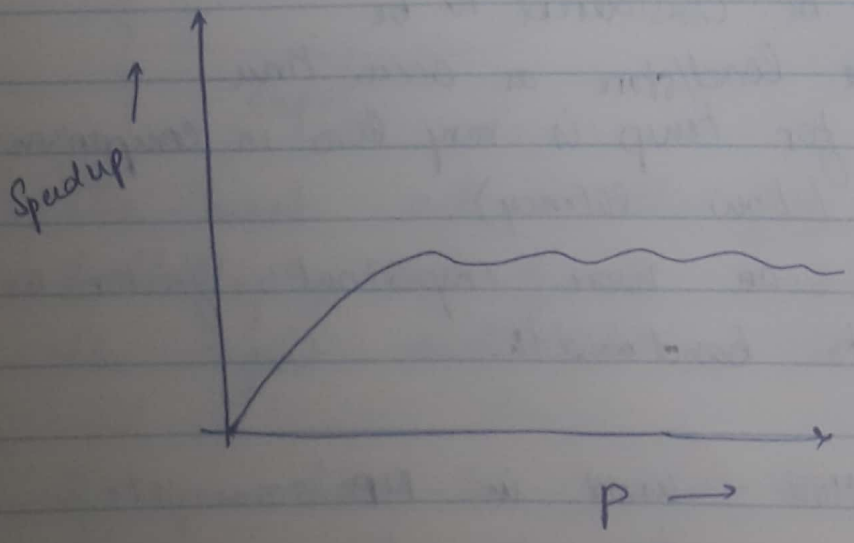
1) Amdahl's Law → possibility of parallelization
2) Gustafson's Barsis → evaluation of problem
3) Karp flat Metric → Investigate
4) Isoefficiency Analysis → Scalability

Restriction on performance:
- Parallel Overhead
- Serial part

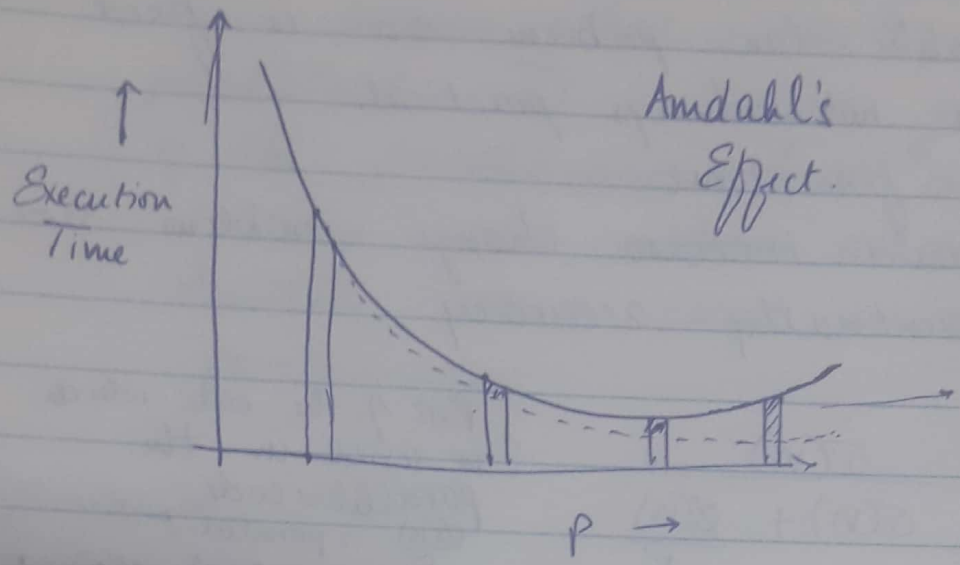
Parallel Part Time = $\phi(n,p)$
Serial Part Time = $\sigma(n)$
Problem size = $n$
Processors = $p$
Overhead = $k(n,p)$

$$\text{Speedup} = \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + k(n,p)}$$

Parallel Time = $\phi(n,p)$




In terms of Amdahl's law, speedup saturates. as it does not take communications into account.




Amdahl's Effect.

This is the time required for communication. The other part is actual processing time which eventually saturates

- Thus, there is always an optimum number of processors for any ~~ever~~ problem

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{No. of processors}}$$

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

The denominator has a term of $P \cdot \sigma(n)$. $\sigma(n)$ proportionately decreases as we increase problem size. Thus, proportion of serial part decreases.

$$\text{Speedup} \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{P} + k(n,p)}$$

$$\leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{P}}$$

- For Amdahl's law, problem size is fixed
  This is not always practical.
  Instead, fix time.
  This would ~~increase~~ change problem size and eventually accuracy.

$$S_1 = \frac{\sigma(n)}{\sigma(n) + \frac{\phi(n)}{p}}$$

, Part of the code which is serial. in the parallelized code. $\frac{\phi(n)}{p}$ is parallel part

$\sigma(n) + \phi(n) = 1$

$$1 - S_1 = \frac{\phi(n)/p}{\sigma(n) + \frac{\phi(n)}{p}}$$

$$\text{Speedup} = \psi(n,p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p}}$$

$$= S + (1-S)p$$

$$= p\text{et } S + p - Sp$$

$$\leq p + (1-p)S$$

p = no of processors
S = serial fraction

$$\therefore \boxed{\psi(n,p) \leq p + (1-p)s}$$ Gustafson's Law.

- Only the initial assumptions are different.
  Time taken for execution here is fixed
  ie $\sigma + \frac{\phi}{p}$. Problem size can vary

This assumes that the code can be parallelized. It is used for evaluation of the parallelization. how accurately it was done within a certain time. Amdahl talked about possibility of parallelization.

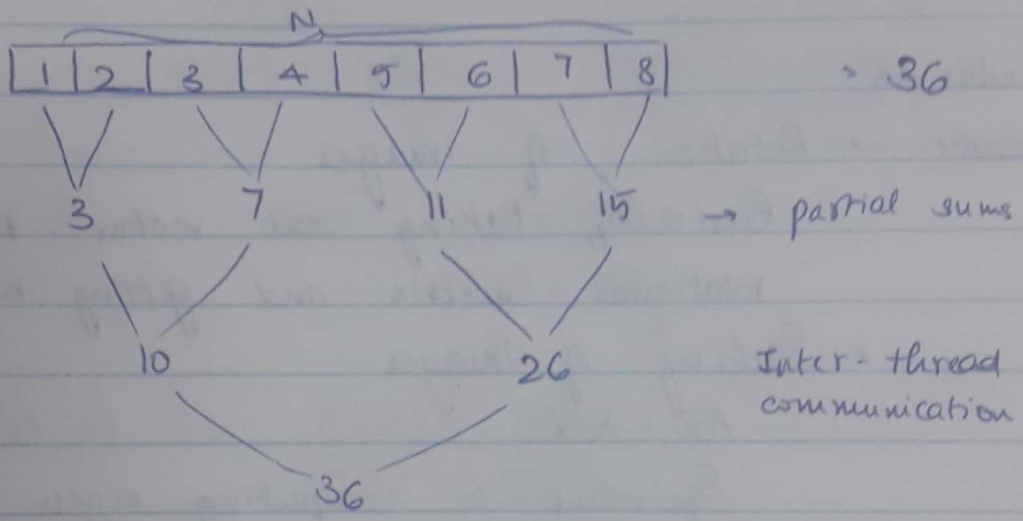
September 13
TUESDAY.

- Rotation of images.
  Basically, taking one vector, & applying rotational matrix and getting new vector.
- Scaling of images
  $A\vec{x} = \lambda\vec{x}$
  Equivalent to computing eigen values.
- Rotation of images is similar to convolution. masking which can be used in machine learning for feature extraction.

- Random variables must be
  PS. is Completely ~~unrelated~~ independent
  (ii) Equally probable.
  PI computing for using random numbers uses probabilistic methods to get a deterministic answer.
  Look at how rand() is applied in C
- Cycle of random numbers generator. The

numbers will begin repeating after a point. So, cannot give the same seed (initial condition) to each thread as they would begin generating the same random numbers.

- Reduction addition




There is some concept known as step complexity. Even if you have infinite processors, cannot compute in a single step.

Here, Steps = $\log_2 N$ N = size of array.

Operations at each step = $\frac{N}{2}$ (1st step) + $\frac{N}{4}$ (2nd step) + ... + 1

$= \frac{N-1}{2-1} = 1 + 2 + \ldots + 2^{(\log_2 N - 1)}$

$= \frac{1\left(1 - 2^{(\log_2 N)}\right)}{2-1} = N - 1$

## Amdahl's law

- $f = \frac{\sigma(n)}{\sigma(n) + \phi(n)}$
- Serial in serial code
- Problem size is fixed
- Scaled speed up

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

- Does not consider overhead

## Gustafson's law

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

- serial in perallel code
- Execution time is fixed
- Scaled speed up.

$$\psi \leq p + (1-p)s$$

- Does not consider overhead

- Karp Flatt Metric

| p | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $\psi$ | | | | |

Include in report.

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

Idle time wasted — overhead — overhead functions — Time taken to finish in single processor

The wasted time is scaled by this as is divided among the processors

For the serial part, only one processor will work on it and rest will be idle. Thus, time wasted $= (p-1)\sigma(n)$

$\sigma(n)$ = serial time

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

$$(p-1)\sigma(n) + pk(n,p)$$

$$= p\sigma(n) - \sigma(n) + pk(n,p)$$

$$= pT(n,p) - \phi(n) - \sigma(n)$$

$$= pT(n,p) - T(n,1)$$

$$(p-1)T(n,1)\,e = pT(n,p) - T(n,1)$$

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

$$= epT(n,1) + (1-e)T(n,1)$$

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

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

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

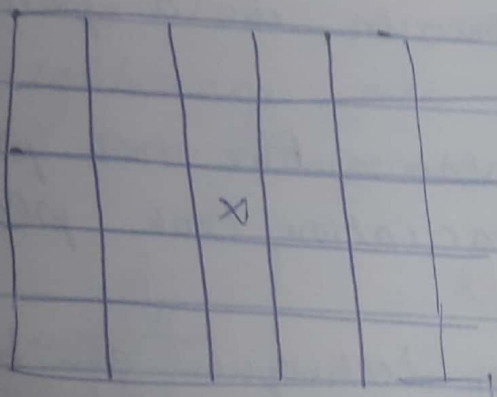
$$\boxed{e = \frac{1/\psi - 1/p}{1 - \frac{1}{p}}}$$

September 20, TUESDAY

eg. Cellular automata → logic → physics
→ Engineering Problems

Conway's Game of life → evolution of Society

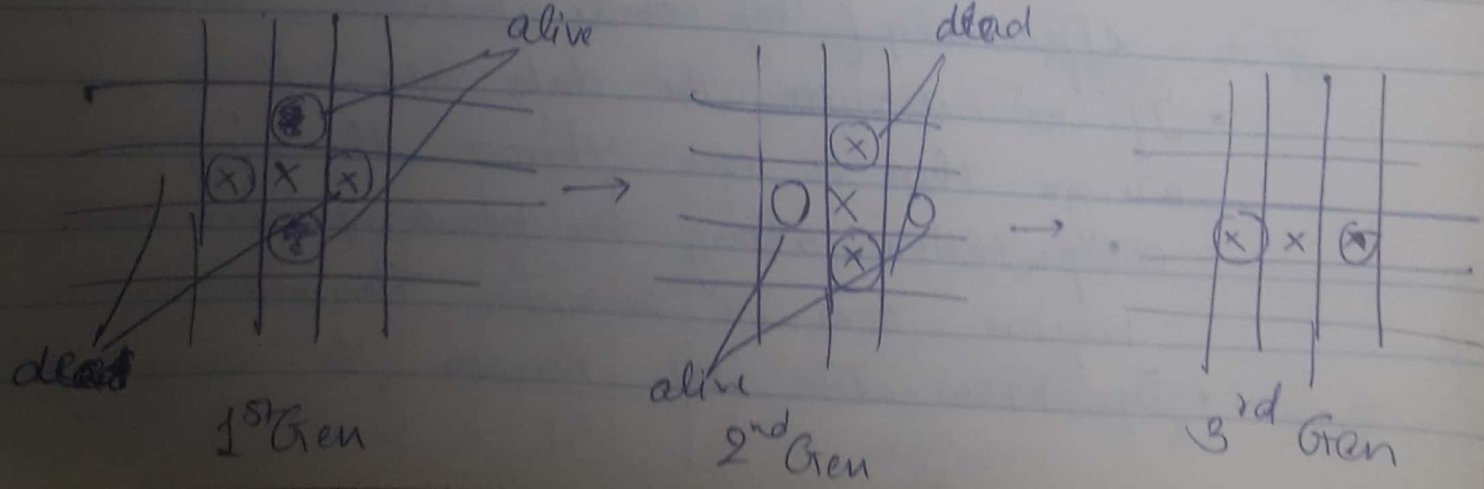
- Any social problem is dynamic in nature in both space and time
  - 2-D domain
  - States - sit, stand, lie
    → States may have rules / may evolve



Size of problem can be controlled by no. of generations we go with the problem to continue.

Size of problem does not remain in the 2-D domain.

States of the problem → live, die



This process will continue in an oscillatory system.

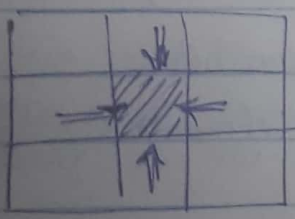
- Application:

(i) Studying pattern
(ii) Evolution of society - Society may have started at some location & then shifted to some other location (also became a bigger comm.)
(iii) How to spread rumours in a particular scenario?
(iv) Location in a place - Fire took place. How will evacuation took place?

- Which parallelization techniques can be used?
  - Every step (in the next generation) depends on the previous step.
  - Spatial locality will be used because we are looking at neighbours
  - Data from adjacent cells can be used effectively (by comm. between threads)
  - If we divider the data into blocks, assign these to each thread, then for computation along the boundary we will require comm. between threads
  - Synchronization bet. threads is essential

before moving to the next generation.
(Load balance / load imbalance)
• Load Balance - Same no. of FLOPs are done by each

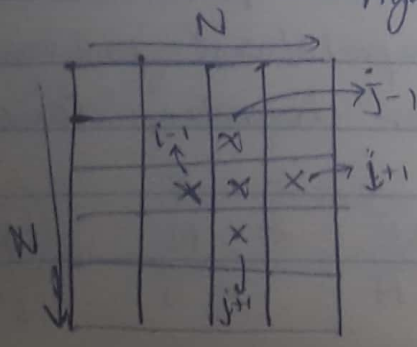
eg. Simulating Heat transfer.
Differential equation.
Application: Heat flow in processes
Heat flow depends on the gradient of temperature.
If we take a cell and look at the neighbouring cells if they are at higher or lower temperature.



We should stop when there is no heat flow. Steady state.

$$T_{new} = T_{old} + k\{(T_{top} - T_{old}) + (T_{bottom} - T_{old}) + (T_{left} - T_{old}) + (T_{right} - T_{old})\}$$

$$T_{new} = T_{old} + k\{T_{top} + T_{bottom} + T_{left} + T_{right} - 4T_{old}\}$$




September 21, WEDNESDAY.

$$\Delta\varphi = \frac{\partial\varphi}{\partial t}$$

The behaviour of water/gas diffusion

The problem will try to reach the steady state (based on law of nature). This depends on the neighbourhood.

$$\frac{\partial \phi}{\partial t} = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = \Delta \phi \text{ (space)}$$

(time)

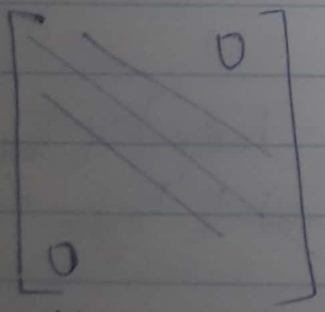
Try to vectorize the equation $Ax = B$

$$= \frac{\phi(i+1,j) + \phi(i-1,j) - 2\phi(i,j)}{(\Delta x)^2} + \frac{\phi(i,j+1) + \phi(i,j-1) - 2\phi(i,j)}{(\Delta y)^2}$$

System of linar equ → Put it in a matrix → $x = A^{-1}B$ → Solve!

use matrix multiplication

Engineering & mathematical problems → diagonal / tridiagonal elements are present → Sparse matrix.



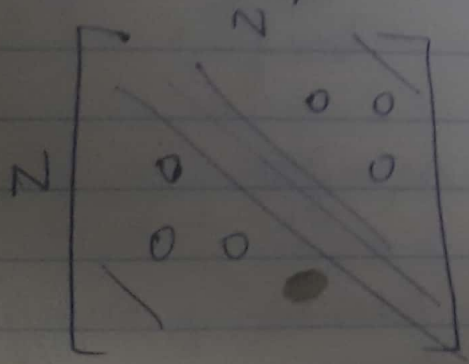
Matrix multiplication

Sparse matrix using data compression.

Data compresiin will increase the possibility of the data in cache.

Such sparse matrix can be compressed.

Computational problem generally have boundary condition.




$\frac{N^2}{5N} \rightarrow \frac{N}{5}$ Reduce by a factor of $5N$

→ Compression of Matrix:

$$\begin{bmatrix} 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{matrix} 4 & 5 \\ 0 & \\ 2 & 1 \\ 1 & \end{matrix}$$

## Compress Sparse Row (CSR):

elements → [4 5 2 1 1]

Column → [2 3 0 1 2]

row → [0 2 2 4 5] → show the amt of memory the first two arrays

(2-0) (2-2) (4-2) (5-4)
= 2 = 0 = 2 = 1

(elements in 1st row) (2nd row) (3rd row) (4th row)

row_no. × row_length + col_index

But that cannot be used as the matrix is sparse. All element-position cannot be used (because row length is not fixed)

The last index-element shows no. of elements present in the compressed matrix.

Length of element array = no. of elements = length of column array

$$\begin{bmatrix} 6 & 0 & 4 & 5 \\ 7 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

row → [0, 3, 4, 6, 7]

↳ Tot. no. of elements in sparse matrix.

Length of row array = N+1 → N×N.

eg. Generating image → using complex numbers (Mandelbrot Set)

↳ Evolve complex numbers to generate images.

Size of image → radius changed.

Change

September 23
FRIDAY

## Scan (Prefix Sum)

$A = [a_0, a_1, \ldots a_{n-1}]$

$Scan(A) = [I, a_0, (a_0 \oplus a_1) \ldots (a_0 \oplus a_1 \oplus \ldots \oplus a_n)]$

I - identity for the operation.

eg 3 1 7 0 4 1 6 3 , ⊕

0 3 4 11 11 15 16 22

Don't need to keep.

In sparse matrix, the third operation is this.

A little like reduction but here, the dimensions of the result are the same as the input.

The final total gives you the length required to store all the values.

~~Don't need to~~

Can overwrite the output onto the same array.

Difficult to parallelize due to dependence on the previous value.

Inclusive Scan:

3 4 11 11 15 16 22 25

Includes the last element.

Exclusive Scan

Include identity and not last

- Can go from inclusive to exclusive and vice versa?

→ Reduction:

- For reduction, number of ~~steps~~ computations is still the same. No. of steps is much less.

Work complexity → work efficient

(Time) Step Complexity → step efficient.

- Follows the simple law of Divide and Conquer

No. of active cores halves at each step.

→ ~~Scan~~

→ Scan.

$$\text{No. of active cores} = \frac{N - 2^{(\text{stride size})}}{2}$$

$$\text{Step Operations at each step} = N - 2^i \qquad \Sigma\left(\frac{(N-2^i)}{2^i}\right)$$

$$= \left[\sum_i (N - 2^i)\right] \log_2 N \qquad = N \log N - \log N$$

$$= N\left(\frac{1\left(\left(\frac{1}{2}\right)^{(k+1)} - 1\right)}{1/2 - 1}\right) \qquad = N\left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \cdots \frac{1}{2^k}\right)$$

$$= N\left(\frac{-1/2N + 1}{1/2}\right) \qquad - \log N.$$

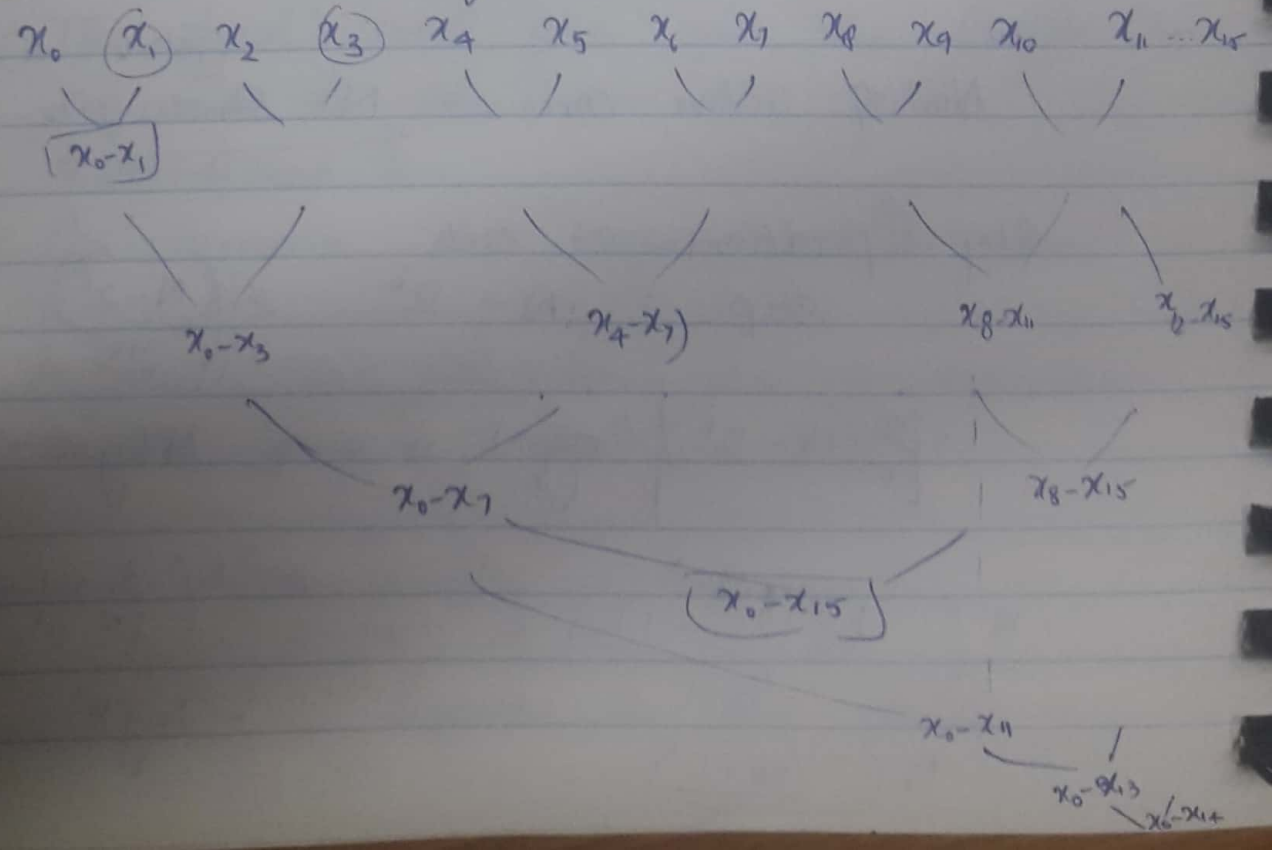
September 21, TUESDAY

Reduction is a pattern, not necessarily always a sum. "Reduction tree"

Reverse Reduction:

This is used to parallelize scan. After using reduction once, you have a lot of partial unused sums stored in the array.
Through this, by created (almost-) a reverse reduction tree i.e repeated same steps in opposite direction order, you can compute all the scan intermediate values.

[Store sums at highest order index]



Scope of optimization:

(i) Processor
(ii) Memory
(iii) Data path

- Computer To Memory Ratio
  eg From Simple to block matrix multiplication we increased this ratio.
  Basically, try and reuse memory.

Three type of problems:

(i) Throughput
  Not necessarily much of data access but lot of computation eg image generation
(ii) Server / & Bandwidth
  More interested of bandwidth is high Communicating with other processor
(iii) Computational Science
  Both.

In all three, we are basically hiding latency.

- Analytical speedup of Reduction

$\left(\frac{n}{\log n}\right)$

Overhead = Cost of Parallel − Cost of Serial

$T_o = pT_p - T_s$

NOTE: Calculate efficiency as $\frac{T_s}{pT_p} = \frac{1}{1 + T_{over}/T_s}$ in assignments

Speedup of Scan = $\left(\frac{n}{n\log n}\right)$

So, no speedup.

Reduction

| | | |
|---|---|---|
| Step | → log n | $\frac{n}{\log n}$ |
| work | → n | $\frac{n}{n}$ |
| Cost | → n log n | $\frac{n}{n\log n}$ |

Not cost optimal. Work / Step optimal

September ~~October~~ 28, WEDNESDAY

Cost of parallel code = (No. of processors) * (No. of steps for parallel code).

We assume that n = p. But not always so.

So, look at efficiency

Efficiency = $\frac{\text{Speedup}}{\text{no. of processors}}$ = $T_s/\text{cost}$ = $S/p$

Cost optimal if Eff. > 1

Cost = $T_s$ / Efficiency

Eg. Speedup = $T_s/T_p$ = $n/\log n$

Cost = $n(\log n)^2$

Efficiency = $1/\log n$

$T_{serial} = n\log n$

$T_{parallel} = (\log n)^2$

→ Not cost optimal by a factor of log n

Now, for p processors, $p < n$

$T_p = \frac{n(\log n)^2}{p}$

Each processor takes $T_p = \log^2 n$ time then for time for $n = n\log^2 n$.

Time taken by p processors = $\frac{n\log^2 n}{p}$

$$\text{Speedup} = \frac{p}{\log n}$$

This shows that speedup decreases if problem size increases. Hence, not cost optimal.

Scaling Down: Assume n processors for n problem size ~~but assume~~ total Then reality is p processors, but still assume total n. (Pad virtual processors which are sitting idle)
So, each processor has to do more computations by factor of $n/p$

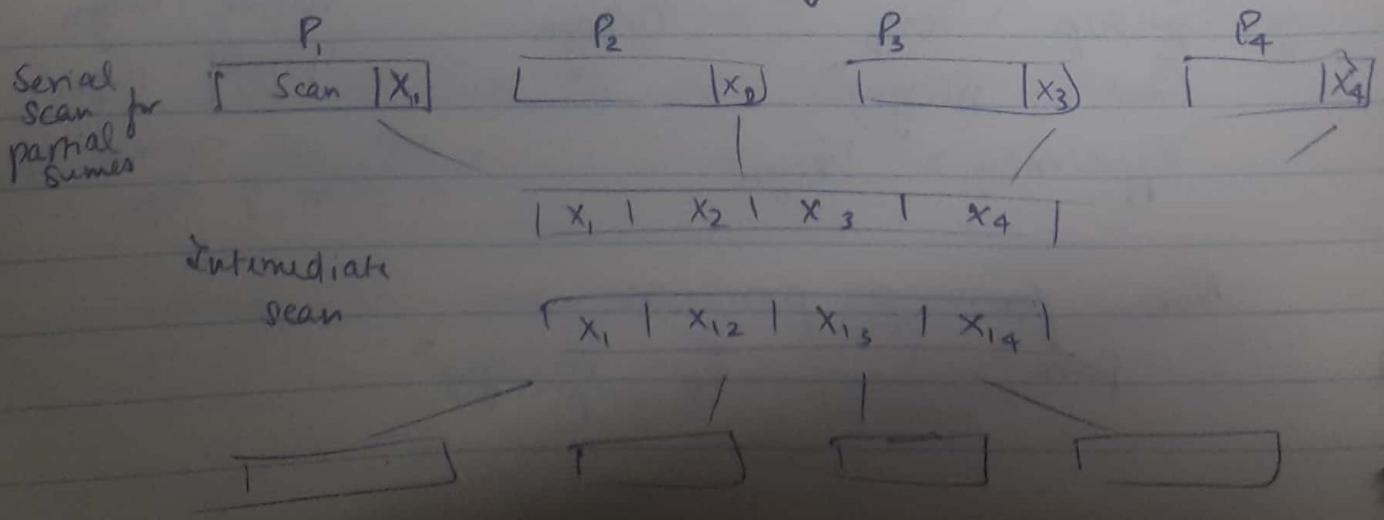
For reduction, $p < n$

$$O(n/p \log p) \sim (n \log p)$$

Step 1: Serial Time $= n/p$

Step 2: Adding partial sum $= \log p$

$$\therefore T_p = O\left(\frac{n}{p} + \log p\right) \sim O(n) \quad p << n$$



September 30,
FRIDAY.

$$E = \frac{T_s}{PT_p}$$

$$E = \frac{1}{1 + \frac{T_o}{T_s}}$$

| Processing Elements (p) | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|
| Speed up | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| Problem size inc | 2 | 3 | 6 | 8 | ... | | 80 |

Overhead / communications increase with increase in problem size.

$\sigma(n)$ is the cost of part that cannot be parallelized. So, this is done only by one processor and rest are idle then.
$K(n,p)$ overhead for each processor.

$$\therefore \text{Idle time} = (p-1)\sigma(n) + pK(n,p)$$

This is a major reason why speedups don't come up to theoretical speedups.

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

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

e { constant
  { increasing.

- If e is constant, with inc. in processors, it shows that the $\sigma(n)$ factor is the restricting factor.
  This is because $k(n,p)$ depends on p and but its effects (though it increases) is not reflected in e.
  $\sigma(n)$ only depends on n.
- If e is increasing, $k(n,p)$ affects.

→ Scalable Parallel Systems: Isoefficiency:

Cost optimum algorithms have efficiency of $O(1)$

{ No. of processors
{ Problem Size

If you reach a part where e becomes constant, you cannot optimize further because it means that you are going into serial part and that cannot be touched.

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

$$= \frac{p(\sigma(n) + \phi(n))}{\sigma(n) + \phi(n) + \underbrace{(p-1)\sigma(n) + pk(n,p)}_{\text{Tot. overhead.}}}$$

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

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

$$= \frac{1}{1 + \frac{T_o(n,p)}{\sigma(n) + \phi(n)}} \quad \& \quad \frac{\text{parallel overhead}}{\text{Serial cost}}$$

$$\therefore \epsilon(n,p) \leq \frac{1}{1 + \frac{T_o(n,p)}{T_o(n,1)}}$$

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

$$T_o(n,1) \geq \left(\frac{\epsilon(n,p)}{1-\epsilon(n,p)}\right) T_o(n,p)$$

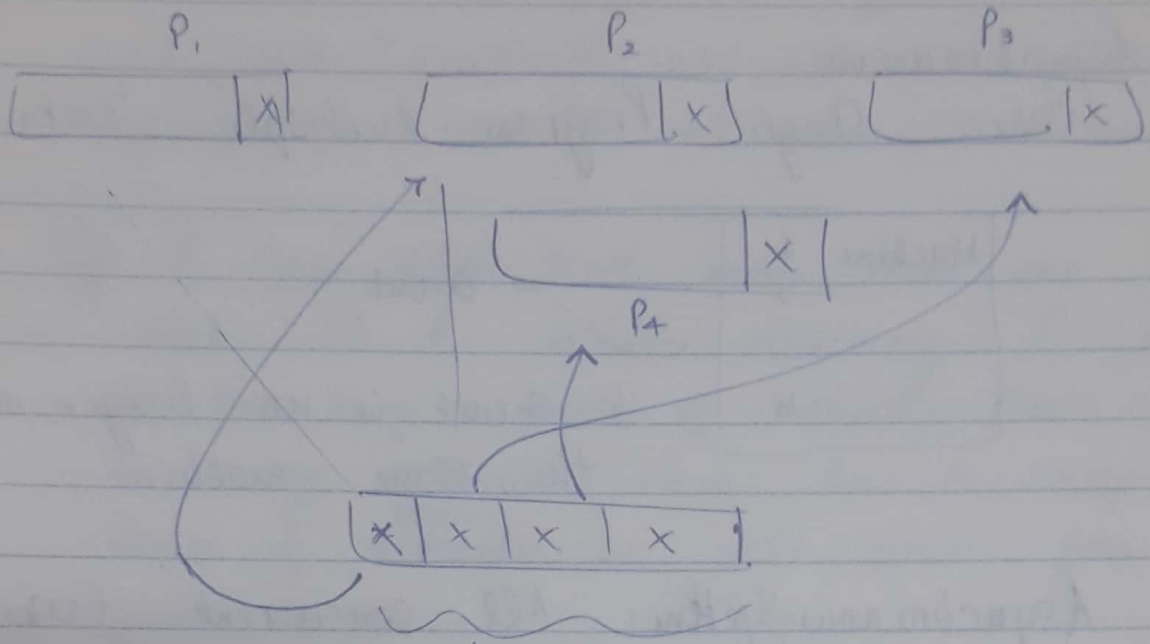
We want to keep this ratio constant. ie we want efficiency to remain constant.

$T(n,1) \geq$ constant $T_o(n,p)$

To keep it constant, we ~~have~~ must increase the problem size along with the number of processors.
With only increase in processors, efficiency would decrease.

There are other factors that affect efficiency as well. Memory access, cache organization, etc.

Thus, Scan and Reduction are designed only for equal size-processors.
If size is more, idle time increases.
Thus, reduce problem size to no. of processors first and then apply reduction on that.




Apply scan here. on last elements.
Now, add each of the ~~equivalent~~ elements to corresponding threads subarrays to get the complete scan. Can used exclusive scan.!

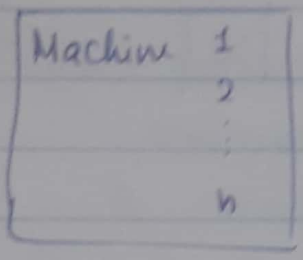
October 4, TUESDAY (i) Shared Memory Parallel Computers
(ii) Distributed - Memory " "

OpenMP cannot be used or distributed.
So, use MPI (

In shared, you create a.out and then the OS directly takes data.
In distributed, you create a.out and then explicitly send it to each system.

Asynchronous

SPMD: Single Program Multiple Data




a.out is no longer on the same machine.

Asynchronous System: All concurrent tasks execute asynchronously.

| $P_0$ | $P_1$ |
|---|---|
| a = 100; | receive (&a, 1, 0); |
| send (&a, 1, 1); | printf("%d\n", a); |
| a = 0; | |

We want $P_1$ to receive a = 100 and not a = 0. So,

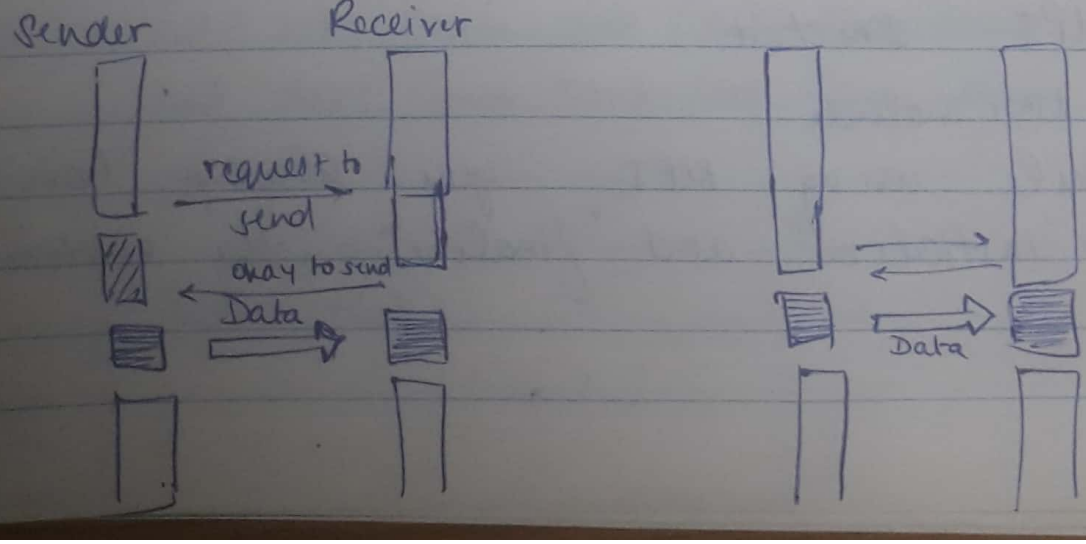
- One method is to wait for $P_1$ to receive a before $P_0$ advances to a = 0 command.
- This causes lots of overhead.
- We can use buffered space instead, Copy a into another variable b and send b. Thus, $P_0$ can manipulate a with any issues.

no. of elements size

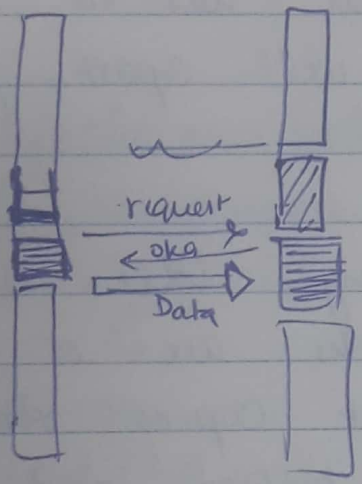
send (void *sendbuf, int nelems, int dest)
receive (void *recvbuf, int nelems, int source)

- If $P_0$ can also send more than one data to $P_1$. While receiving, $P_1$ has to keep a track of which data is being received. Thus, has to keep a track of order as well apart from just source.

- In non-buffered blocking, idle time increases and deadlocks are a big issue
- In buffered, the sender copies the data into the designated buffer and returns after the copy operation has been completed. The receiver can also use the buffer and will use the data only when required by it.
  Idling overheads decrease but copying overheads increase.



Second case has minimum idling time.
However, this is very unlikely to happen because the two processors are independent and run asynchronously.
In the first case, sender has to wait for the receiver to be ready.




Here, receiver needs some information but the sender cannot send yet. This leads to idling time for receiver.

→ Buffered Blocking Message Passing Operations:

Whether buffering works depends on hardware as well

→ MPI structure.

MPI include

- While using MPI, you always have to "initialize" and "finalize" the environment

MPI_INIT(& argc, & argv) ← command line arguments. Same as main()

MPI_Finalize( )

↳ No other MPI call can be made after this.

"You have several processors for each process but each process can have multiple threads."

→ Communicators and Process Groups:

↳

- Communicators are groups of processes that can communicate with one another.

eg Student can be part of HPC group and DBMS, etc. However, don't have to be part of all.

- Defines a communication domain. Can only communicate in this domain.
- One process can belong to many different communication domains.

int MPI_Comm_size (MPI_Comm comm, int * cpus)

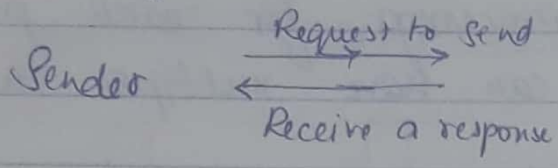
int MPI_Comm_rank (MPI_Comm comm, int * rank)

cpus ← no. of

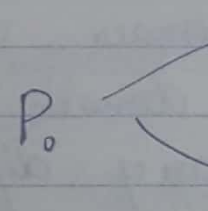
rank ← process within that

October 7, FRIDAY

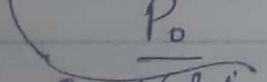
- Save time in communication
- Correct information is obtained at the receiver's end.
- Communication is enabled (done correctly) using hardware.




- Buffer can reduce idle time.
  Blocking Mechanism - guarantee that data will be transferred correctly
- Overlap communication with computation.
  But it is essential that we take guarantee that data required to be transferred should not be overwritten.




- Send data to a process anonymously is possible in MPI.
- So, we will not be able to find if the data is correct or not (and 1 element (can be specified from which processor) whether int, float)



| $P_0$ | $P_1$ |
|---|---|
| send(&A, 1, 1) | send(&A, 1, 0) |
| tag 1 | tag 1 |
| Receive (&B, 1, 1) | receive (&B, 1, 0) |

- If we use non-blocking then it will lead to dead-locking. Because the 'send' from $P_0$ should be received by $P_1$
- But $P_1$ rather than receiving, sends data to $P_0$. So, now $P_1$ is blocked as $P_0$ is waiting until data is received from $P_1$ (which is not sent)
- If we use non-blocking send-receive method then the code will function

Deadlock.

$P_0$

:

Explicit communication ← send(&a, 1, 1)

:

receiv(&b, 1, 0)

:

$P_1$

:

receive(&a, 1, 0)

:

If instead, the function is to dead-lock always this may lead in blocking state

- Non-blocking mechanism
  - Have to ensure that the update is done correctly
  - Use buffered space. So, fast & without incorrect code.
- CPU doesn't necessarily have to take part in the communication.
  Sender → sending process waits till a copy of data and stores it in the buffer.
  Idling time is completely gone by using buffer.
  Place it in the buffer of receiver
  Placed in the buffer ? hardware. Communication buffer

(i) Blocking buffered transfer.

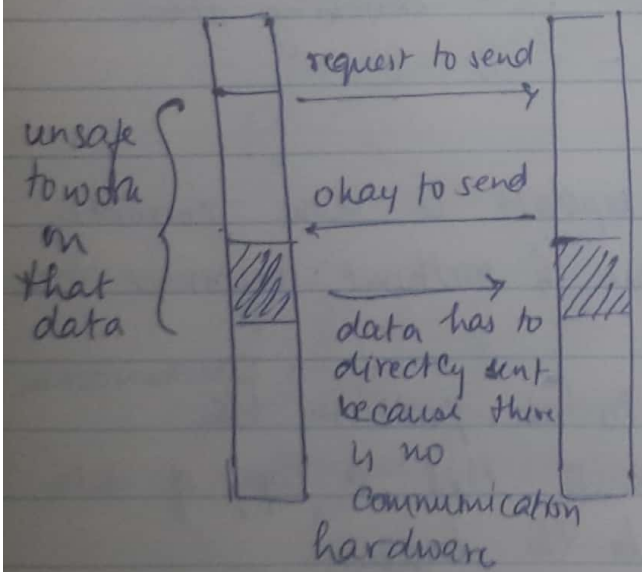
- In the presence of communication hardware with buffer at the receiver and sender's end.

- Buffer is supported by hardware
- Sender only copies data into its buffer and the receiver can access data at any time

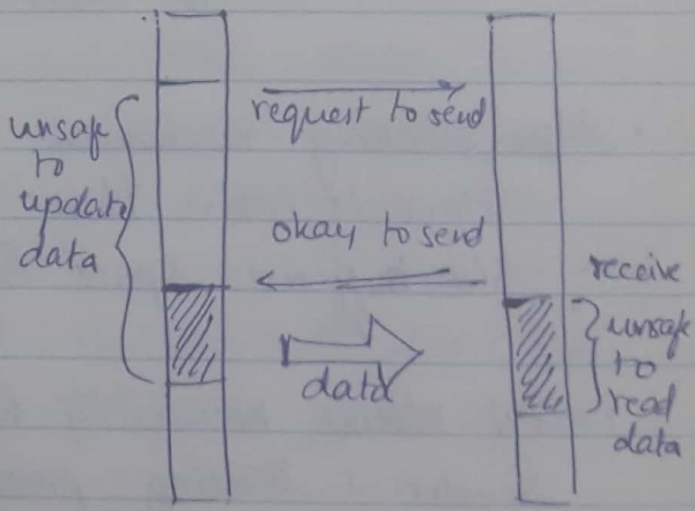
- Buffer is not supported by hardware
- Sender obstructs the receiver from continuing & copies data in its buffer.

Non-Blocking Non-Buffered:

(a) In absence of communicator hardware




(b) In presence of hardware




Communication hardware → Non-buffered space where it stores data

Simply ensures the CPU does not need to overlook communication. Assigns to hardware & continues its computation.

October 12, WEDNESDAY

Several processors run each process. Thus, each has to 'initialize' and 'finalize' MPI environment respectively. So, everything comes within that.

MPI_COMM_WORLD : All processors are under this communicator.
You can communicate between two processors only if they are part of the same communicator.

MIMD: Multiple executable files on different processors.
However, we will mostly be working with SIMD. ie One shared file

A process can be a part of several communication domains.
You can also have unicast or multicast.

- Buffered - non-buffered
- Send-receive / Isend, etc
- Type of communication.

→ Communicators:

From no of cpus, no of processors in domain and current cpu id, it can get what the part of the code that cpu is supposed to do.

eg MPI_Send( buffer, length of (buffer), data type,

only length of data to be comm.

rank of comm. procs, tag, comm. domain)

to identify send-recv pair

MPI_Recv( ... , & status)

Always blocking!

Helps with error handling like size/length mismatch

Non-blocking receive is unsafe.

There is no 'status' argument on the send side. Other functions like MPI_Success which shows this.

MPI_Send ("a")

a[0] = 1.

This statement will not execute unless the entire data of 'a' is sent.
Thus, we can check status.

Point-to-point communication
Collective " : Faster than point-to-point. However, can be used in only some cases.

Status' is an object (like a stack) rather than a simple variable.
Thus, much data can be stored there.
Used as arguments for a lot of other functions.

→ Collective Calls:

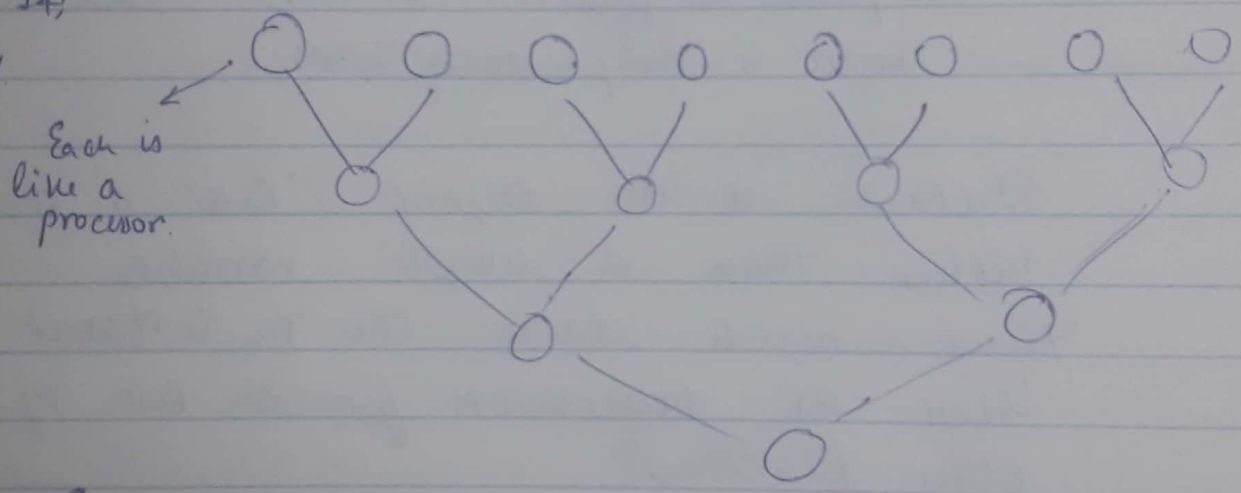
Synchronization is done about through 'barrier'.

MPI_Barrier() → Explicitly synchronize among processors within that comm. domain

MPI_Bcast: Broadcast to all other processors. All processors call MPI_Bcast with same root and communicator arguments.

* Look at all the function, return types and see if they can be used as arguments for elsewhere.

October 14;
FRIDAY




Since there are dependencies on the previous lines, regardless of how many processors you have, you cannot complete it in one step. This is called <u>Span</u>

Here, the degree of concurrency is 8. This is because we treat each node as an independent processor.
We have 8 nodes (or 8 data points) so

each of them could have executed independently. That would lead to 8 processes. Don't get stuck with reduction. The tree signifies any parallel computation. [Reversing the tree makes it easier to see the 8 processes. 1 process is eventually divided into 8]

Now, on reduction. Threads = p

i) Work = 7 (no of computations)

Work Law: $\boxed{pT_p \geq T_1}$

ii) Cost = (num_t) log (num_t) Steps = log(num_t)

iii) Span / Depth / Steps = $\log_2 n$ if $T_\infty = \infty$

OR

Critical Path length (Shortest Time)

iv) Speedup = $\frac{T_1}{pT_p}$

Step Law: $\boxed{T_p \geq T_\infty}$

If we have p processors, no. of steps taken

would be atleast steps taken by ∞ processors

If $n > P$,

$$T_P = \frac{n}{P} + \log_2 P$$

$$T_\infty = \log_2 n$$

$$\frac{T_1}{P} \leq T_P \leq \frac{T_1}{P} + \log_2 n$$

Brent's Graham's Law:

$$\frac{T_1}{P} \leq T_P \leq \frac{T_1}{P} + T_\infty$$

$\frac{T_1}{P}$ signifies when there is a single step of computation.

RHS ie the upper bound in takes span into consideration. That assumes a unit time of computation at each step and so, adding.

v. Iso efficiency.

At what rate should you increase your problem size ($W$) so that the efficiency remains constant.

$$W = K T_0(P, W)$$

$K \propto E$

For reduction, $T_o = p \log p$ $\quad p \geq w$

[Overhead at each step for each processor]

If $p \longrightarrow p_1$

$$\frac{w_1}{w} = \frac{p_1 \log p_1}{p \log p}$$

$$w_1 = \left(\frac{p_1 \log p_1}{p \log p}\right) w$$

$$\frac{w}{w_1} = \frac{p^2 + p \log p + w^2 p^4}{p_1^2 + p_1 \log p_1 + w^2 p^4}$$

Not always possible to write the iso-efficiency equation for all problems. May be that problem size is not exactly scalable in that way.

eg. Matrix addition : $W^2$ problem size.

eg. $W = K(p^2 + p \log p + w^2 p^4)$

Problem size should be increased based on the asymptotic upperbound of overhead.

NOTE: If strictly talking theory, you can assume no. of processors to be equal to problem size

eg $T_s = n\log n$ $C_1, C_2, C_3, C_4$ → 4 algorithms

| | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|
| processors | $n^2$ | $\log n$ | $n$ | $\sqrt{n}$ |
| $T_p$ | 1 | $n$ | $\sqrt{n}$ | $\sqrt{n}\log n$ |
| Speedup | $n\log n$ | $\log n$ | $\sqrt{n}\log n$ | $\sqrt{n}$ |
| | [$C_1$ is most optimum by this] | | | |
| Efficiency | $\frac{\log n}{n}$ | 1 | $\frac{\log n}{\sqrt{n}}$ | 1 |
| | [$C_2$ & $C_4$ are most optimum here] | | | |
| Cost | $n^2$ | $n\log n$ | $n\sqrt{n}$ | $n\log n$ |
| | [$C_2$ & $C_4$ most optimum here] | | | |

Overall, $C_2$ is the most optimum.
$C_4$?

How do you ~~get~~ check ~~optir~~ minimum time with minimum no. of processors?
ie minimum time and cost optimal.

~~Tp=~~

$$T_p = \frac{n}{P} + \log_2 P$$

$$\frac{dT_p}{P} = -\frac{n}{P^2} + \frac{1}{P} = 0$$

$$\therefore P = n$$

Thus, $p=n$ gives minimum time.

$$\text{Cost} = p \times \log p$$

$T_p = \log p$

No. of steps = time taken in parallel

$$\epsilon = \frac{s}{p} \qquad \therefore p = \frac{s}{\epsilon}$$

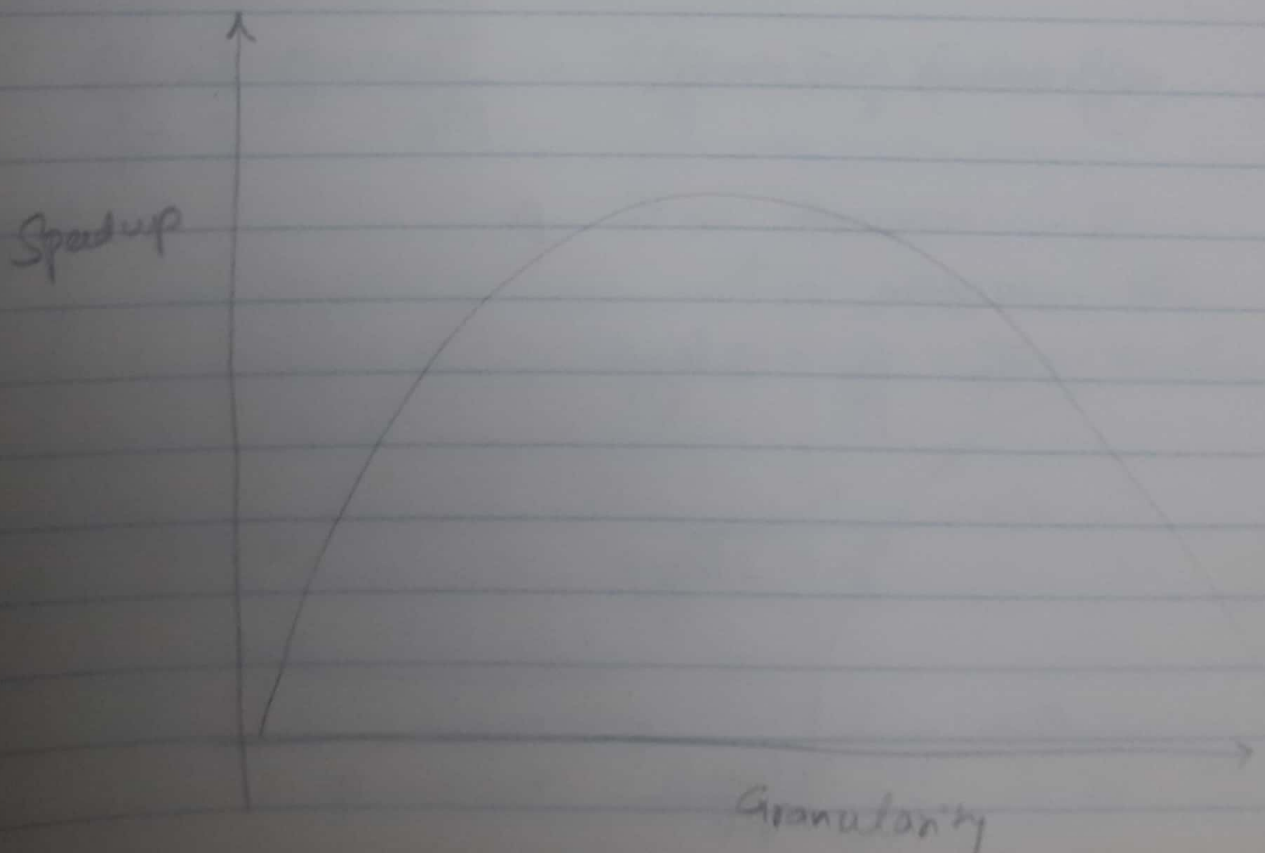
$$C = \frac{s}{\epsilon} \times \log p$$

$$= \frac{\frac{T_s}{T_p}}{\epsilon} \times T_p$$

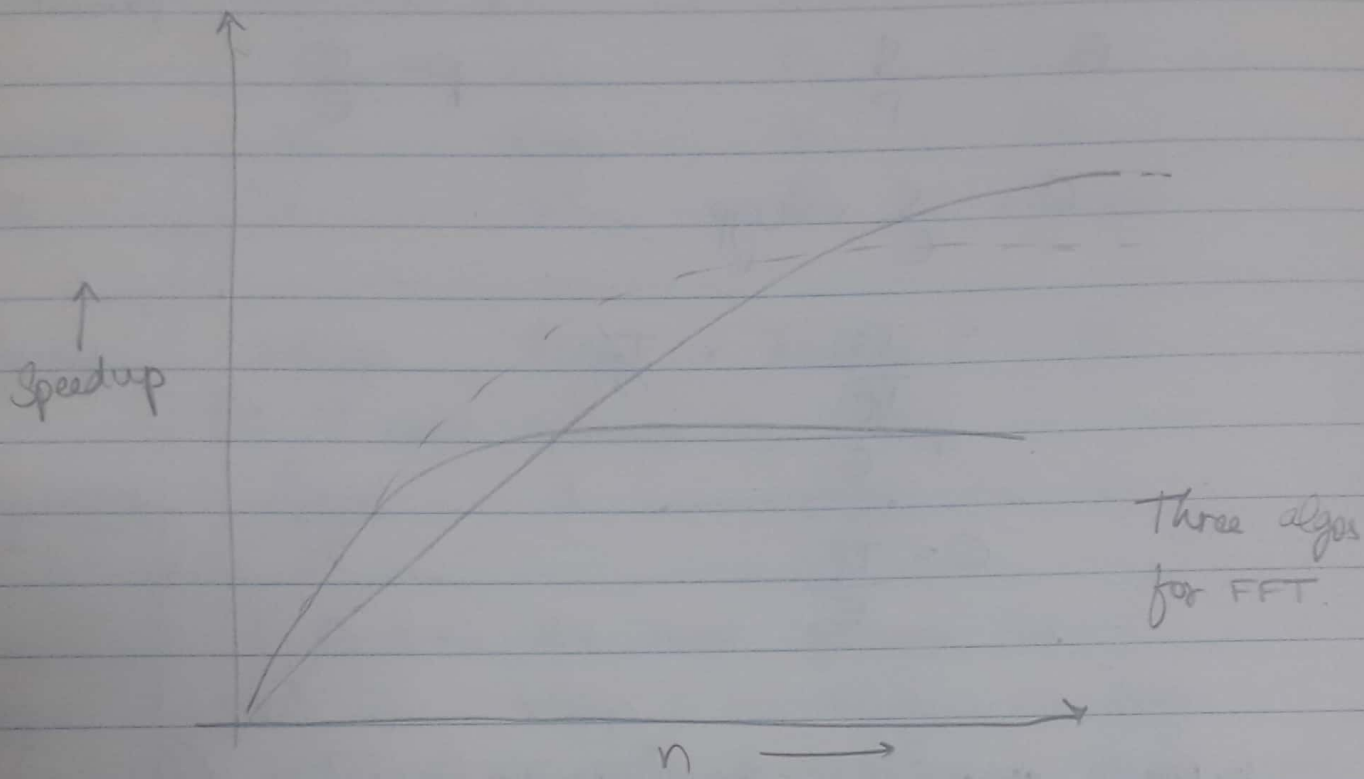
$$C = \frac{T_s}{\epsilon}$$

September 30
FRIDAY

Effect of Granularity on Performance



# Scalability of Parallel Systems




Increasing cost with increasing n.

$$\text{Efficiency } E = \frac{S}{P} = \frac{T_S}{PT_P} = \frac{1}{1 + \frac{T_0}{T_S}}$$

$\therefore T_0$ increases with p.

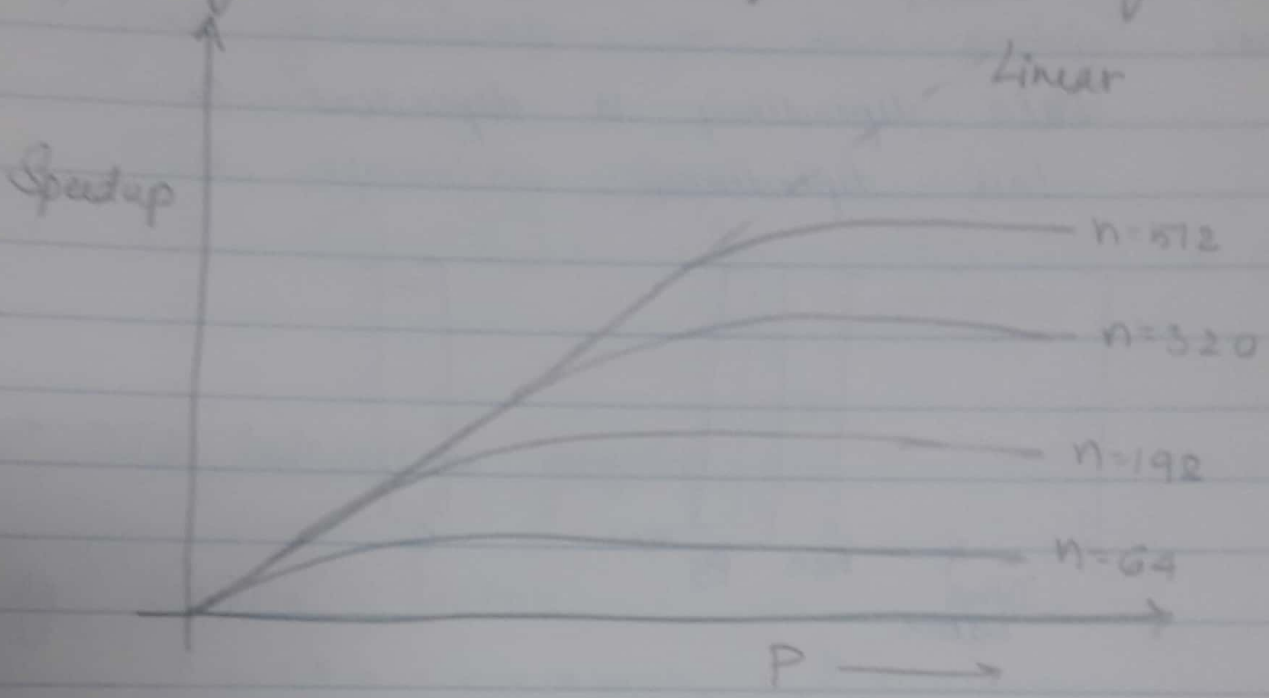
For reduction,

$$T_P = \frac{n}{P} + 2\log p$$

$$S = \frac{n}{\frac{n}{P} + 2\log p}$$

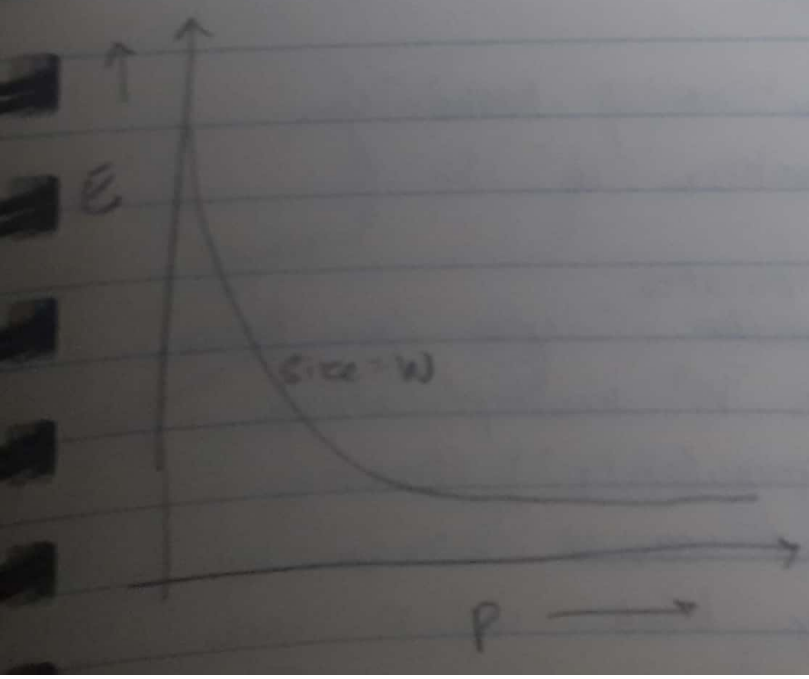
$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$

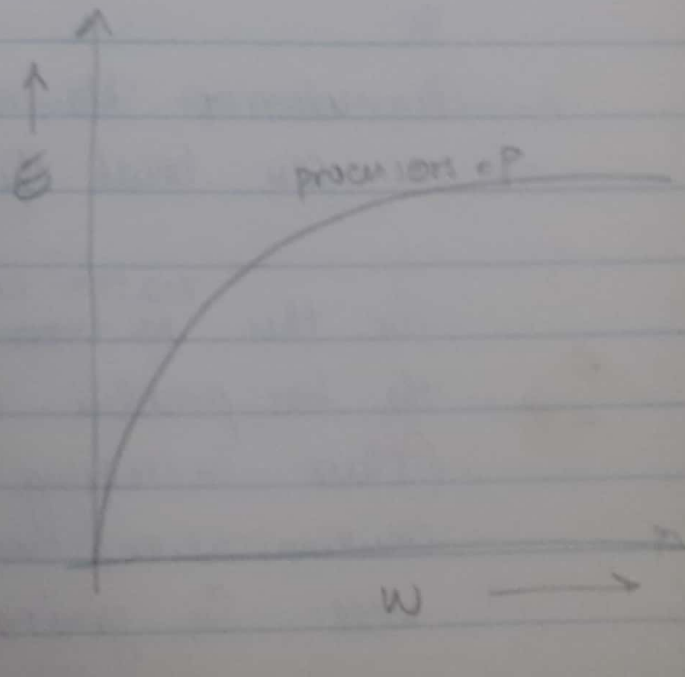
## Scaling Characteristics of Parallel Programs




When assumed p=n, instead of linear curve, we get saturating curve

## Isoefficiency - Metric of Scalability

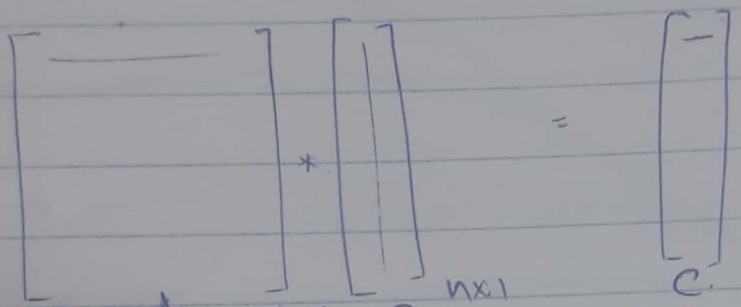


October 21, FRIDAY.

## Decomposition, Tasks and Dependency graphs:

Data dependency is dependent on task dependency.




Each row of A will require entire B. There is no task dependency since each element of C can be calculated individually.
However, data dependency exists.

- You will want to increase granularity to make it more efficient.

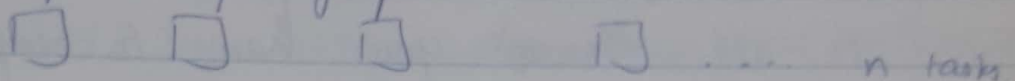
%0

Granularity - Maximum no. of tasks you can break the problem in.

For this matrix multiplication, you can break the ~~tot~~ problem into $n^2$ multiplications. (Thus increasing granularity?) However, communication overhead would increase as well. So, generally break in n tasks.

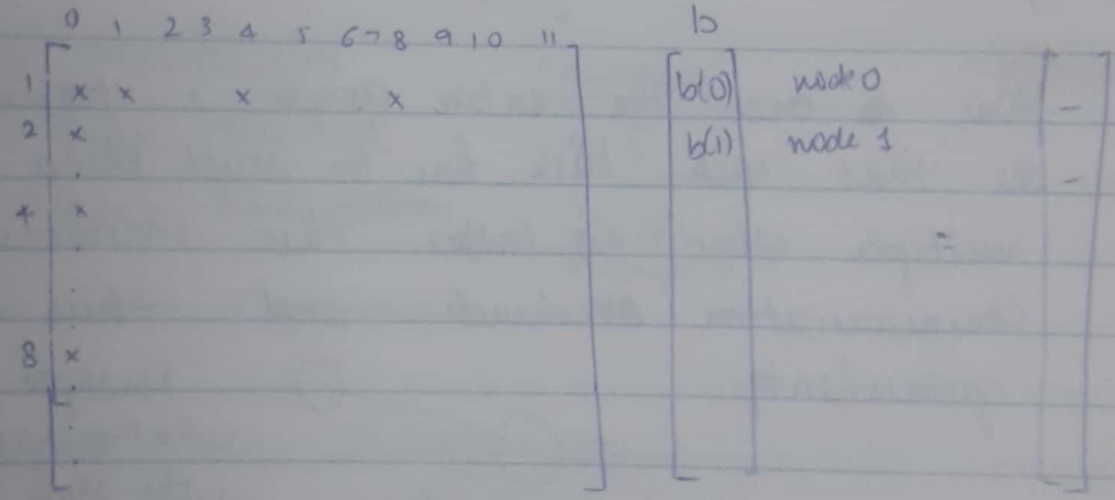
Thus, there is an int inherent bound on how fine granularity for each problem.

Task dependency graph.

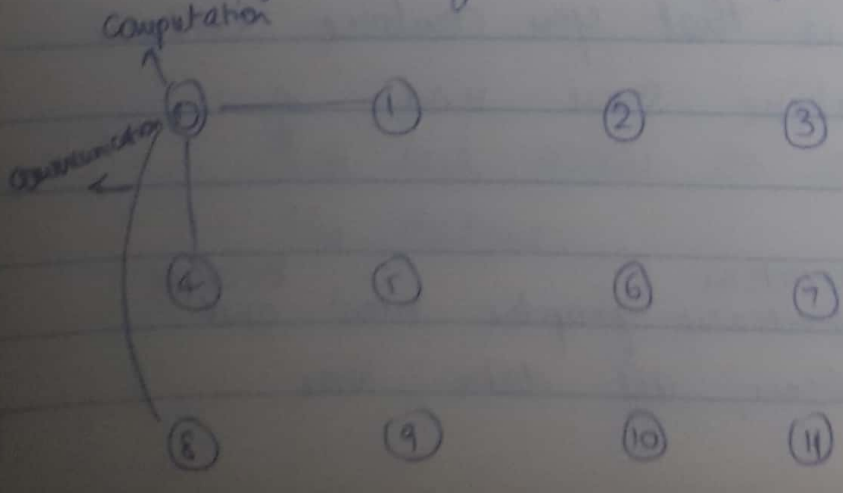



→ Task interaction graphs:

eg Sparse Matrix



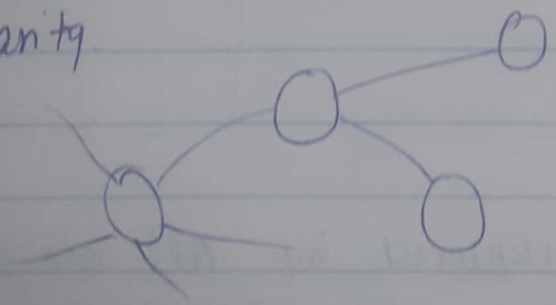

b[0] is required by all non-zero elements of the first column of A.



- These graphs are independent non directional. Only shows the interaction between two tasks. Who sends or receives doesn't matter.
- [In distributed memory, b init available to all. Hence, you have to explicitly send it to other processors through MPI]

Task dependency graphs are inherently directed. Each level is computed only when the previous has been computed

Now, once the entire graph is formed, you see that each task has to send data to multiple other no. tasks. This increases communication overheads and thus reduce granularity



Multiple nodes interact with the same node. Can combine.

One solution is that you combine certain tasks. ie combine some nodes of the graph

(Talk about interaction dependency graphs now because and on shared systems, all data was

available to everyone. Not true in MPI)

→ Processes and Mappings:

- Determined by both dependency and interaction graphs.
  Task dependency → for load balance.
  Task interaction → communications reqd

↓ | ↓ | ↓

Data decomposition
eg.. Image processing
Matrix Multiplications
(Most ours assignments)

Recursive
eg. $P_i$
Divide & Conquer.

Speculation.

Output Data Decomposition.
eg. Matrix multiplication.

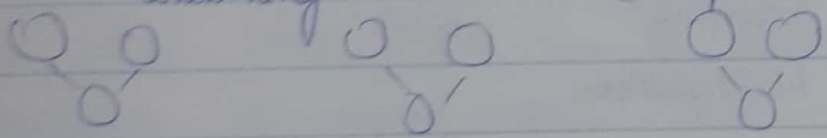
$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

Each task works on a its part of the output.

Input Data Decomposition
eg Image filtering.

Each task only works on its part of the input.

Intermediate
eg. Block multiplication.
Compute partial sums from each blocks
This is input decomposition.
Then, compute the actual output by combining these. (output decomposition)



$AX = B$.

Very memory intensive
Avoid taking $A^{-1}$ because if A was a sparse matrix, the inverse would become dense.

October 25, TUESDAY 13, In-SEM 2.

Called loop unrolling.

```
do i = 1, n-1, i+=2
    b(i) = a(i) + s* a(i+1)
    b(i+1) = a(i+1) + s* a(i+2)
end do
```

3 accesses for 4 comp instead of 2 for 2.

0

0 250000

(1×25000+1) + 25000

25000

sum

#progma

for (1, $10^7$):

sum += f()

$$4\left(1 - \frac{1}{3} + \frac{1}{5} \ldots\ldots\right)$$

parallel_measurement.c

parallel_script.c