AKSHAT SARASWAT

HPC

Instruction Level Parallel Processing

→ Earliest use of parallelism in designing PE to enhance processing speed, was pipelining.
  ↳ extensively used in RISC which was succeded by superscalar processors
    ↳ execute multiple instructions in one cycle.

→ The idea of ~~using~~ in superscalar processor is to use parallelism available at instruction lvl by increasing no. of arithmetic & functional units in PE.

→ This idea is further exploited in VLIW processors where one instruction word encodes more than one operation

Pipelining of Processing Elements

→ ~~pipeling~~

→ pipelining uses temporal parallelism to ↑ speed of processing.

→ Pipelining is an effective way if certain "ideal" conditions are satisfied:

① Possible to break instruction into no. of independant parts, each taking equal time to execute

② There is locality in instruction execution. Means that instructions are executed in order they are written. If there is 'jump around', pipelining not effective.

③ Successive instruction can use work done by prev. instruction & are independant.

④ Sufficient resources available.

Fetch → Decode → Execute → Access Memory → store in Register
FI DE EX MEM SR

→ However, these ideal conditions not always satisfied:

① Not possible for each part to have same execution time. However, delays can be added to make them equal.

② Real programs have branches & the execution is not always sequential. It is however possible to predict when branches will be taken a little ahead in time.

③ Successive instructions not always independent.

④ There are always resource constraints in processor as chip size is limited.

Calculating Avg. Clock Cycles per Instruction

Non-pipelined case

$$\frac{\sum (\%\text{ instruction of type } p \times \text{clock cycle needed per instruction of type } p)}{100}$$

Pipelined case

(Ideal)

Total no. of instructions executed = m

No. of clock cycle per instruction = n

Time taken w/ no pipelining = mn

Time taken w/ pipelining (ideal) = $n + (m-1)$

$$\text{Speedup} = \frac{mn}{(m-1)+n} = \frac{n}{n/m + (m-1)/m}$$

if $m \gg n$ then speedup $\approx n$

(Non-ideal)

These cases occurs due to fact that extra buffer registers are introduced b/w pipeling stages & use of these registers, lengthens the clock cycle. Instead of 1, it takes (1+e) cycles.

$$\text{speedup} = \frac{mn}{n + (m-1)(1+e)} = \frac{n}{(1+e)}$$

Ques) A non-pipelined comp uses 10 ns clock. The avg. no. of clock cycles per instruction required by this machine is 4.35. When it is pipelined, it needs 11 ns clock. Find speedup.

Ans Assume,

depth of pipeline = n

no. of instructions = N

speedup = $(4.35 \times 10 \times N) / ((n+N-1) \times 11)$

since $N \gg n$

$$\text{speedup} = \frac{4.35 \times 10}{11} = 3.95$$

Delays in pipeline execution

pipeline hazards → delays in pipeline execution of instructions due to non-ideal conditions.

non-ideal conditions:

- Limited resources
- Successive instructions are not independent
- Programs have braches & loops.

Delay due to Resource Constraints (Structural Hazard)

→ Delayed due to unavailability of resources when required during execution of an instruction

→ If common memory is used for both data & instructions, only one can be carried out, other has to wait. Forced instruction waiting is called pipeline stall.

→ May also be delayed if one step of execution takes longer than one cycle.

└ To solve this we can use extra hardware

→ Loss of speedup due to resource non-availability

Assume a floating point division & integer addition.

Total instruction = m

Fraction of floating point inst. = f

Assume floating point inst. takes $(n+k)$ clock cycles each.

Time taken w/ no pipelining = $m(1-f)n + mf(n+k)$

" " w/ pipeling = $n + (m-1)(1+kf)$

$$\text{Speedup} = \frac{m(1-f)n + mf(n+k)}{n+(m-1)(1+kf)} \approx \frac{n}{1+kf} \qquad \begin{matrix} m \gg n \\ n \gg kf \end{matrix}$$

If lets say 10% of Inst. are floating point & take 2 extra cycles to execute. $k=2$, $f=0.1$

speedup = $0.833\,n$ $\Rightarrow$ 16.6% loss in efficiency.

Delay due to Data Dependency (Data Hazard)

→ Delayed due to the fact that successive instructions are not always independant of one another.

→ Results produced by an inst. may be needed by succeeding inst. & results may not be ready when needed.

→ Eg:

```
ADD R1, R2, R3
MUL R3, R4, R5
SUB R7, R2, R6
INC R3
```

→ To avoid data hazard, there are 2 methods:
hardware technique → register forwarding
software technique

hardware technique (register forwarding)

→ Instead of saving the value in register file, the result of an operation is stored in a ↓ buffer. The next instruction instead of waiting to fetch data from stored register, can fetch data from buffer. A path is provided from buffer to ALU unit. which bypasses MEM & SR cycles.

→ many pipelined processors have register forwarding as a standard feature.

→ This does not eliminate pipeline delay.

software technique

→ it reduces delay & in some cases, totally eliminates it

→ sequence of instructions is reordered w/out changing meaning of prog.

Delay due to Branch Instructions (Control Hazard)

→ A branch disrupts the normal flow of control.

Ques) In our pipeline, maximum ideal speedup is 5. Let % of unconditional branches is 5% & conditional branches is 15%. Assume 80% of conditional branches are taken in programs.

Ans) No. of cycles per instruction (ideal) = 1

Avg. delay cycles due to unconditional branch = $3 \times 0.05 = 0.15$

Avg. delay " due " conditional "

= delay due to taken + delay due to not taken branches

$= 3 \times (0.15 \times 0.8) + 2 \times (0.15 \times 0.2)$

$= 0.45$

$$\text{Speedup} = \frac{5}{1 + 0.15 + 0.42} \approx 3.18$$

% loss in speedup = 36.4%. → essential to reduce this delay.

→ Again there are two ways : hardware, software aids.

Hardware modification to reduce delay due to branches

→ Primary idea is to find out address of the next instruction to be executed as early as possible.

- for eg. adding ALU at certain steps/stage of pipeline may help in predicting the branch.

→ Eg. Assume by adding this ALU hardware, we reduce delay to 1 cycle if branch is taken & 0 if not.

Q) Assume again 5% unconditional jumps, 15% conditional jumps & 80% of conditional jumps as taken.

Ans) Avg. cycle delay w/ extra hardware

$= 1 \times 0.05 + 1 \times (0.15 \times 0.8) = 0.17$

$\text{Speedup} = \frac{5}{1.17} = 4.27$

% loss in speedup = 14.6%

→ This was simple hardware addition however commercial processors are more complex & have variety of branch instructions. Therefore other technologies are used.

→ Two methods both of which depend on predicting the instruction which will be executed immediately after branch instruction.

→ This prediction is based upon execution time behaviour of a program.

→ ① Branch Prediction Buffer
- cheap
- less effective
- uses small fast memory

② Branch Target Buffer
- expensive
- more effective
- big fast memory
- requires more control circuitry.

Branch Prediction Buffer

→ In this technique some of the lower order bits of address of branch instructions in a prog. segment are used as addresses of the branch prediction buffer memory.

→ The contents of each location of this buffer memory is address of the next instruction to be executed if branch is taken.

→ In addition, 2 bits [(00, 01) & (10, 11)] are used to predict whether branch will be taken or not.
(00, 01) → not taken; (10, 11) → taken

→ If the inst. is branch instruction, lower order bits of its address are used to look up branch prediction buffer memory.

→ The bits are examined, if 10 or 11, control jumps to branch address found in branch prediction buffer.

→ why two bits & not one ??
  ↳ single bit predictions are incorrect whereas 2-bit predictor is found cost-effective

⇒ We know if inst. is branch or not at DE step of pipeline

Branch Target Buffer

→ Used at instruction fetch step itself.

→ BTB memory:

Address	Contents	Prediction Bits
Address of branch instruction	Address where branch will jump	1 or 2 bits

→ has complete address of all branch instructions.

→ contents are created dynamically i.e. whenever branch statement encountered, its address & branch target address placed in BTB.

→ Once a BTB entry is made, it can be accessed at inst. fetching phase itself.

→ In case of a loop, when it is executed first time, the branch inst. governing the loop will not be found in BTB but the subsequent iterations will find branch target at fetch stage itself thereby saving 3 clock cycles delay.

Software Modification to Reduce delay due to Branches

→ The primary idea is for compiler to rearrange the statements of prog. in such way that the statement following the branch statement (delayed slot) is always executed once it is fetched without affecting the correctness of the program.

→

Original

ADD R_4, R_5, R_6

ADD R_1, R_2, R_3
JMP X

DELAY SLOT

Rearranged

ADD R_1, R_2, R_3
JMP X

ADD R_1, R_2, R_3

→ If no such statement is available, a No Operation (NOP) statement is used as a filler after branch so that when it is fetched, it does not affect meaning of prog.

Difficulties in Pipelining

→ Some more difficulties in designing pipeline processors:

- illegal instruction codes
- page faults
- I/O calls

} exception conditions

→ In case of undefined instruction, hardware malfunction or power failure we terminate the program.

→ If pipeline processing can be stopped when an exception condition is detected in such way that all instructions that occur before the one causing exception are completed & all which were in progress can be restarted, the pipeline is said to have precise exceptions.

→ Some exception types in computer:

- I/O request
- initiating break point
- Arithmetic overflow / underflow
- Page faults
- Undefined instruction
- Hardware failure
- Power failure.

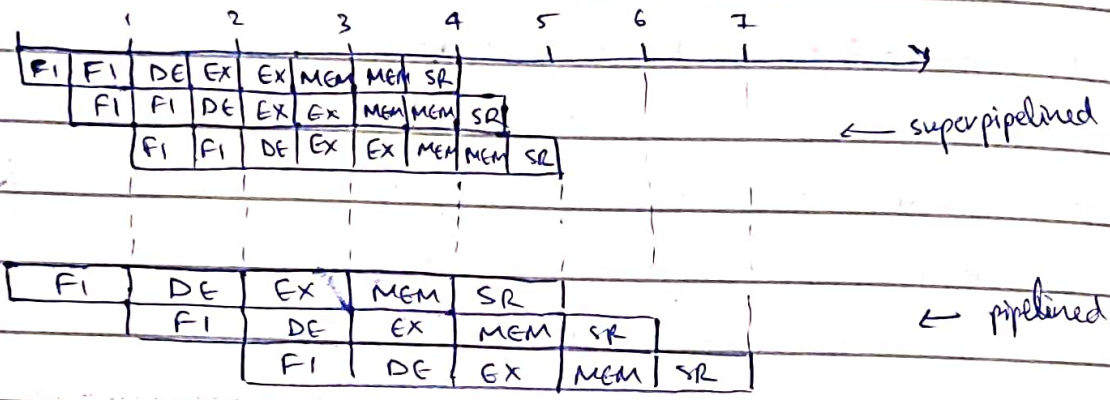
Superscalar Processors

→ In a pipelined processor, we assume that each stage of pipeline takes the same time → one clock interval.

→ Possible for stages to have < 1 clock interval like DE & SR

→ Superpipelining

Divide each clock cycle into two halves/phases



All difficulties of pipeline processing, also occur in superpipeline

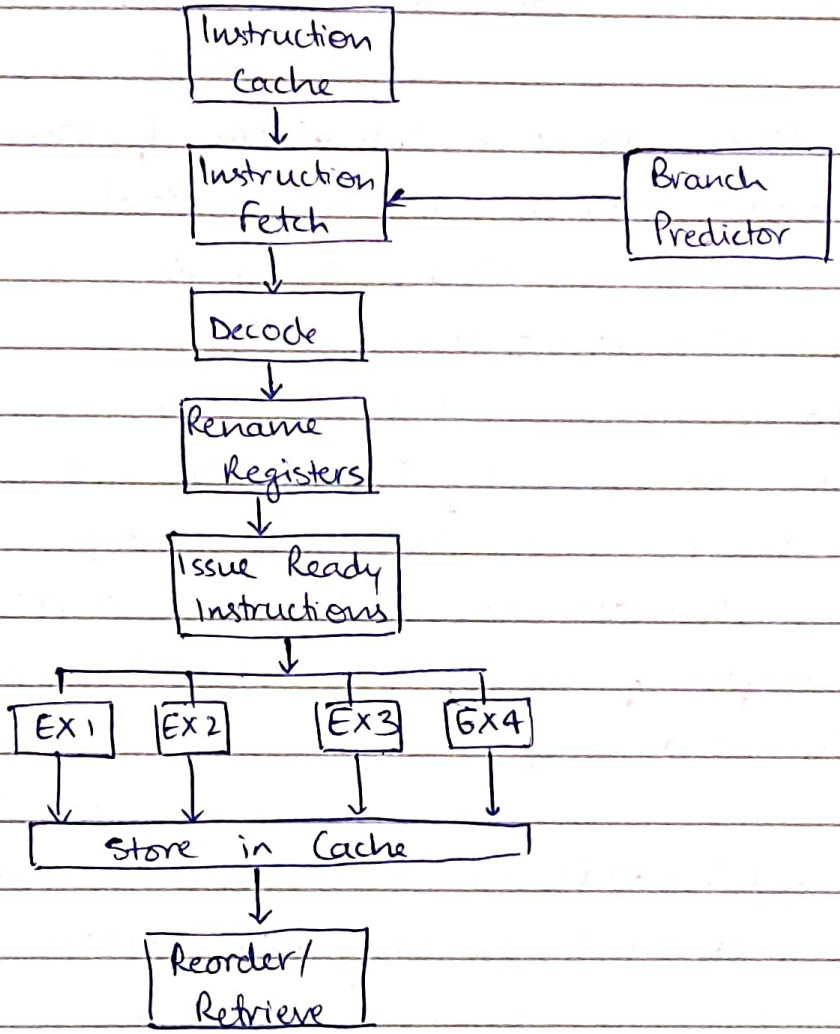
→ Superscalar

- Here we combine temporal parallelism used in pipeline & data parallelism by issuing several instructions from an instruction thread simultaneously in each cycle.
- For superscalar processing:
  - → hardware should permit fetching several inst. simultaneously
  - → data cache must have independent ports for read/write to be used simultaneously.
- Superscalar depends on available parallelism in groups of instruction of program.
- what happens in case of branches? we use same procedure of setting branch prediction bits & deploy BTB

superscalar RISC

superscalar CISC

- Eg: MIPS10000 & Power PC | Pentium 4
- The hardware of superscalar processors provides an instruction buffer (instruction window) w/ capacity of several instructions
- From the buffer, system picks instructions in a way that completion time is minimized.



→ One method to reduce branching while exposing more instruction parallelism is loop unrolling

→ Methods used by hardware to detect dependency:

- → Score board (dynamic superscalar): separate register (n bit register). Each bit is set if register is in use
- → Register Renaming (data dependency): Eliminates WAR & WAW. It abstracts local registers from physical registers.

VLIW Processor

→ Very Long Instruction Word

→ Problems in designing superscalar
- Have to duplicate register, decoder, ALU
- difficult to schedule instructions dynamically to reduce delays

→ Hardware looks only at a small window of instructions. Scheduling them to use all available processing units is sub-optimal.

→ Compilers take a more global POV & rearrange code to better utilize resource & reduce delays.

** → Alternative to superscalar which uses compilers to expose inst. which have no dependency & require diff. resources of processor.

** → In this architecture, a single word, incorporates many operations which are independent & thus can be pipelined w/out conflicts. If such operations are not found, the slot is filled w/ a no-op.

→ Trace scheduling → Based on predicting path taken by branch operations at compile time using some heuristics given by prog.

→ Challenges in designing VLIW:
- Difficulties in building hardware
- Inefficient use of ~~boor~~ bits in a VLIW
- Lack of sufficient inst-lvl parallelism in programs.
- Need much higher memory & register file bandwidth to support wide words & multiple read/write
- Binary code compatibility b/w 2 generations of VLIW also very tough.

ARRAY PROCESSORS

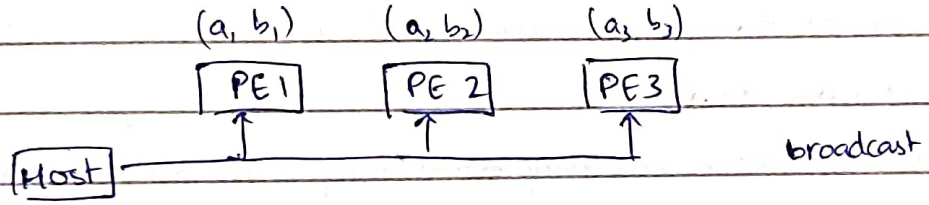
→ A vector comp. adds 2 vectors $(a_1, a_2, \cdots a_n)$ & $(b_1, b_2, \cdots b_n)$ by streaming through pipelined ALU

→ Another method would be array of n PEs. where all work simultaneously. An add instruction is broadcast to all PEs. by host. Such organisation of PE called array processor

→ Array processor use data parallelism

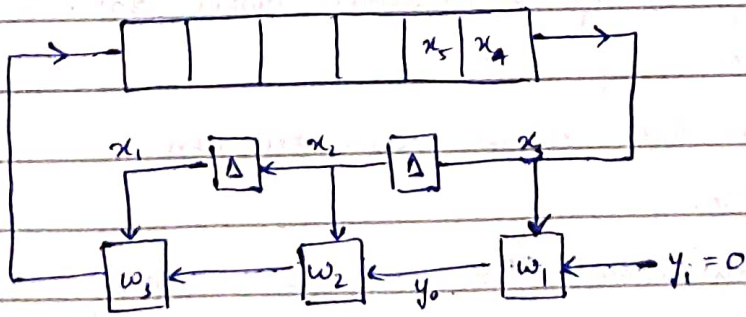
→ If instead of broadcasting inst. from host, inst. are stored in pvt. memory of each PE (now computing elements), the host has to issue a start command.

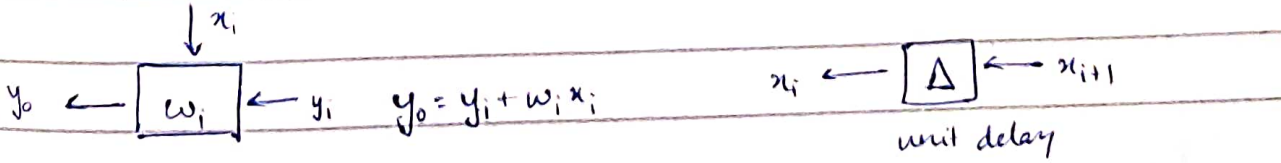
→ This model is called Single Program Multiple Data (SPMD)



SYSTOLIC ARRAY PROCESSOR

→ The general idea of pipeline processing is used in building special purpose high speed VLSI circuits to perform various math operations like matrix mul. & Fourier. Such circuits are called systolic.





AKSHAT SARASWAT

→ primary characteristics of systolic array:
- Flow of data is rhythmic & regular
- Data can flow in more than 1 dimension
- Comms. b/w cells is serial & periodic
- Individual cells only connected to nearest neighbours
- Time ~~for~~ taken for processing by each cell is identical

SHARED MEMORY PARALLEL COMPUTERS

→ Provides global address space
→ Each processor has a pvt. cache
→ Processors connected to main memory either through shared bus or interconnection network.
→ avg. accesstime to main memory from any processor is same.
→ Also called symmetric multiprocessor (SMP)
→ Shared bus is inexpensive & easy to expand.
→ Common prog. & data are stored in main memory shared by all PE

Synchronization of Processes

→ Main prog. creates separate processes for each PE & allocates them along w/ the info on location where data is stored.
→ Each PE computes independently.
→ After completion, they rejoin main program.
→ Two statements:

join : when invoking process needs results of invoked proces

fork : creates a process

→

Process X	Process Y	after forking both run concurrently. At join statement, process X waits till process Y terminates.
⋮	⋮	
fork Y;	⋮	
⋮	⋮	
join Y;	end Y;	

→ lock issue arises

(To get ~~sum~~ sum = f(A) + f(B))

Process A	Process B
⋮	⋮
fork(B)	sum ← sum + f(B)
⋮	⋮
sum ← sum + f(A)	end B;
⋮	
join B	
⋮	
end A	

To solve the issue;

Process A	Process B
⋮	⋮
fork(B)	lock sum;
⋮	sum ← sum + f(B)
lock sum;	unlock sum
sum ← sum + f(A);	⋮
unlock sum;	end B;
⋮	
join B;	
⋮	
end A;	

→ TST (test and set) is also an atomic read-modify-write instruction.

→ Lamport defines sequential consistency as "A multiprocessor is sequentially consistent if result of any execution is same as if the operation of all processors were executed in some sequential order and operations of each individual processor occurs in this sequence in the order specified by its program."

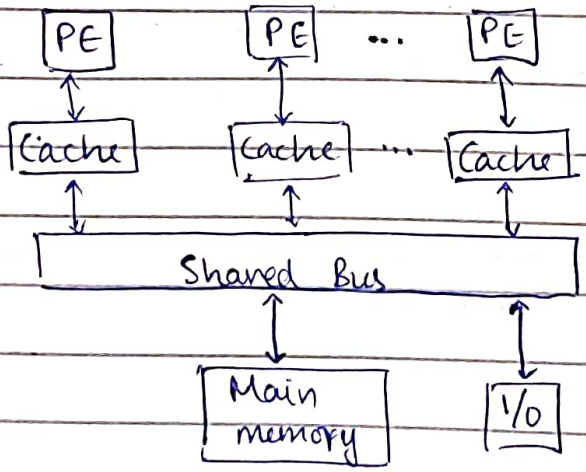
→ In a shared memory computer w/ cache coherence protocol, sequential consistency is ensured.

Shared Bus Architecture

→ In this, each PE has pvt. cache

→ Pvt. cache has 2 functions:

- To reduce access time to data & program
- To reduce the need of all processors to access main memory



→ uses of cache essential but it brings cache coherence problem.

→ Problem arises because when PE writes a data into its pvt. cache in address x, it's not known to caches of other PE. Now if another PE reads data from x of its cache, it will read earlier stored data.

→ There are many protocols to ensure coherence.

Cache Coherence in Shared Bus

→ If there is read request & if block containing data is in cache (Read Hit), it's delivered to processor

→ If data block is not there, (Read Miss) block from main memory containing data replaces block in cache.

→ In case of write request, if block is present it is Write-Hit.
- write-through : updates main memory immediately
- write-later : data written in main memory when the block is replaced from cache.

→ The cache block besides data, has a dirty bit. It is set to 1 when new data overwrites the data originally read from memory.

→ Only when cache block is dirty, should it be written in main memory.

→ Reasons of Read/Write miss:
- Cold miss : when cache is not loaded w/ blocks in working set
- Capacity miss : when cache is too small to accomodate " ".
- Conflict miss : when multiple cache blocks contend for same set in set associative cache.

→ A bus system has advantage that a transaction involving a cache block may be broadcast on bus & other cache listen to it.

→ Thus, cache coherence protocols are based on cache controller of each cache recieving instructions from PE, to which it is also connected & listening to the broadcast on bus. These protocols are called snoopy cache protocols.

MESI Cache Coherence Protocol

→ State diagram representation

→ This protocol invalidates the shared blocks in cache when new data is written in block by any PE.

→ It is write-invalidate protocol, proposed by Intel

→ When new data is written in cache block, it is not written immediately in main memory. In other words, it is write-back protocol to main memory.

→ Defines a set of states in which blocks may be found

→ M (modified) → Data in cache block modified. Old data in main memory. Dirty bit set in cache block.

E (exclusive) → The data in cache block is valid & same as memory. No other cache has this copy

S (shared) → Data in cache block valid & same as memory. Some other may also have copies

I (invalid) → Data in cache block invalidated as another PE has same cache w/ updated value.

MOESI Protocol

→ Proposed by AMD, adds another state 'owned' apart from MESI

→ Main reason for introducing 'owned' is to delay writing modified block in memory which is beneficial in new gen of chips where time to write data in memory >> time to transfer blocks among caches on chip.

O (owned) → Data in cache block valid. Some other cache may have copies. However block in this state has exclusive right to make changes to it & broadcast to other caches.

Memory Consistency Models

→ Consistency definition specifies the rules about read/write & how they alter state of memory.

→ Lamport's sequencial consistency. It implies that order of execution of statements in a program implicitly assumed by programmer is maintained.

→

PE 1	PE 2
X := 0	X := 0
F := 0	F := 0
⋮	⋮
X := 1	p := F
F := 1	q := X

even though this prog. is sequentially consistent, it can give 3 possible results (0,0) (0,1) (1,1)
This is due to data races

→ To get determinat results, make prog. data race free.

PE 1	PE 2
X := 0	X := 0
F := 0	F := 0
B := 0	⋮
⋮	until (B == 1) wait
X := 1	p := F
F := 1	q := X
B := 1	

→ sequential consistency is guaranteed if following conditions satisfied:

- Cache coherence of || computer is ensured
- All PE observe writes to a specified location in memory in same order
- For each PE, delay access to memory until all prev. read/write are completed.

* $X \rightarrow Y$ means X must complete before Y execute

→ In bus based shared memory multiprocessor, seq. consistency is maintained as:

- Cache coherency
- Read by PE is completed in prog. order. If there is cache miss, it delays subsequent reads
- Writes are serealized by bus and observed by all PE in order which they are written.

→ Sequential Consistency requires that operations must maintain $R \rightarrow W$, $W \rightarrow R$, $R \rightarrow R$, $W \rightarrow W$.

Relaxed Consistency Models

→ They relax one or more of above ordering requirements.

→ 3 models:

- Total Store Order (TSO)

  $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow W$ are required

- Partial Store Order (PSO)

  $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$ are required

- Weak Ordering

  $W \rightarrow W$, $W \rightarrow R$ are required

INTERCONNECTION NETWORKS

→ Main advantage is designing the network in such way that there are several paths b/w two modules being connected which increases bandwidth

→ Can be used to interconnect memory to processor & computers

→ Interconnected set of comp. called distributed memory parallel computers.

→ 2 categories:

- Direct: each node connected to neighbour by dedicated communication lines
- Indirect: connected using switching elements

Networks to Interconnect

① Crossbar switch

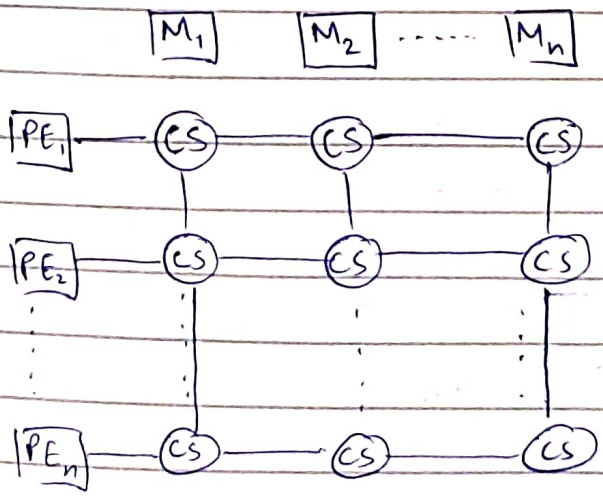
→ allows any processor PE_i to connect any memory M_i

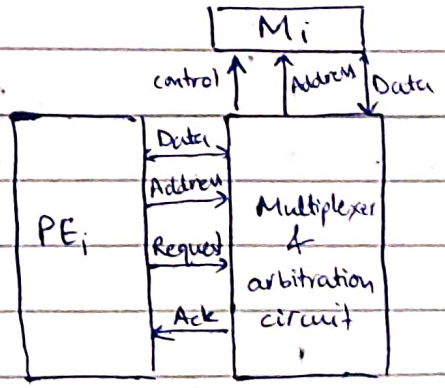
→ allows simultaneous connection of (P_i, M_i) $i = 1$ to n.

→ routing mechanism is called crosspoint switch

→ The switch consists of multiplexers & arbitrators.

→ in case of multiple requests to memory, uses priority queues

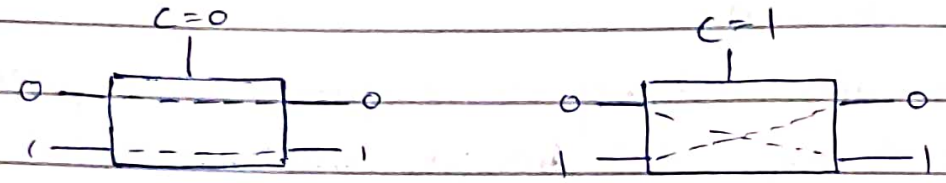




Structure of crosspoint switch

② Multistage inter-connection network

→ Basic component is two-input two-output interchange switch. & control input C.



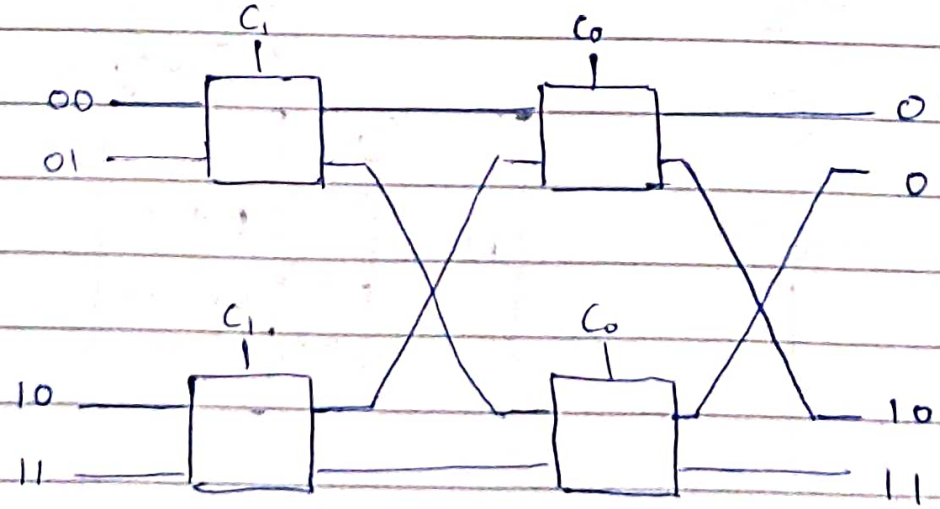
→ A general multistage network has N inputs, N outputs where $N = 2^m$, m = no. of stage

→ For (N x N) crossbar, no. of switches = N^2 whereas for multistage, its $(N/2)\log_2 N$

→ Types of multistage networks:

- nonblocking → crossbar
- blocking
- omega network → uses shuffling connection of $\log_2 N$ cascaded switches
- butterfly network → uses butterfly switch
- benes network → back to back butterfly.

→ Butterfly switch



Direct Interconnection of Computers

→ There are many interconnection networks which connect CE to build parallel computers

→ These methods have advantages like:
- Regularity
- Fault Tolerance
- Good Bandwidth

→ CE are connected through a Network Interface Unit (NIU) to bidirectional switches.

→ Interconnection networks characterized by 4 params:

performance:
- Total Network Bandwidth i.e. total Bandwidth, bytes/second that network can support.
- Bisection bandwidth: determined by imagining a cut which divides network into two parts & finding b/w of links across the cut

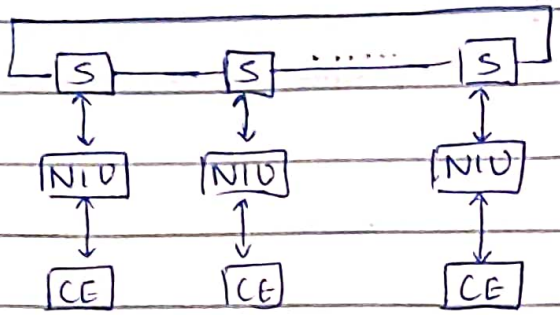
cost:
- Number of ports each network switch has
- Total no. of links b/w switches & switches and CE

→ Types: Ring, 2D grid, toroid, N cube

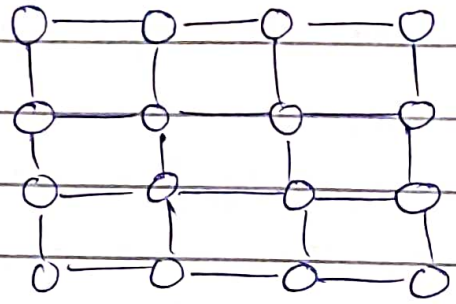
→

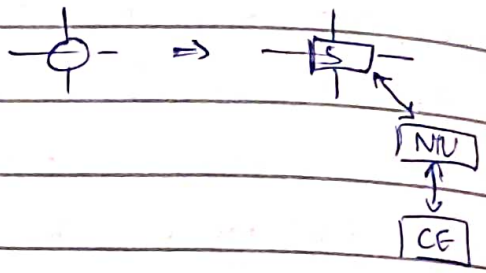
	Bus	Ring	2D Grid	toroid	N cube
Total Bandwidth	B	nB	$2\sqrt{n}(\sqrt{n}-1)B$	2nB	$Bn\log n/2$
Bisection Bandwidth	B	2B	$\sqrt{n}\,B$	$2\sqrt{n}\,B$	$Bn/2$
Total no. of links	–	2n	$3n-2\sqrt{n}$	3n	$n + n\log n/2$
Ports per switch	1	3	5	5	$\log n + 1$

→ Ring

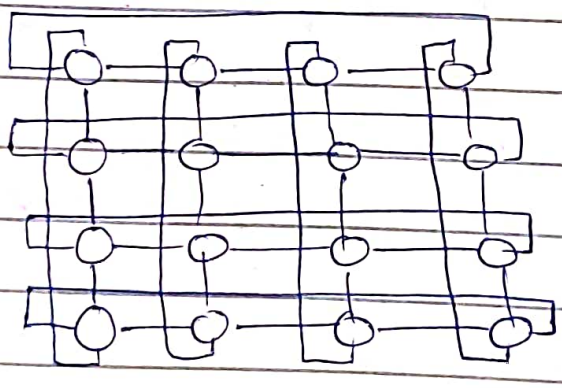


→ Grid

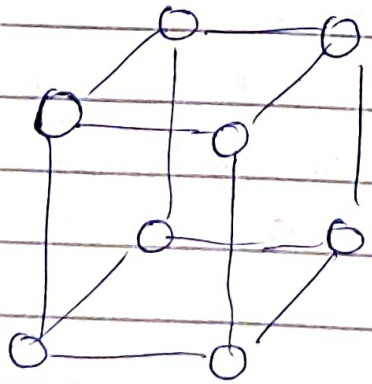




→ Toroid



⇒ Hyper Cube



Routing Techniques for Directly Connected Systems

→ sophisticated switches are routers

→ Routers connect various networks whereas switch connects various devices within a network

→ Routers control transmission of msgs b/w nodes & allow overlapping of computation w/ communication.

→ Main job of router to decide whether incoming msg is intended for CE connected to it or to be forwarded to next neighbour

→ msgs broken into packets which have header & data.

→ Packet switching

messages transmitted b/w nodes using this. Packets leave source & take available route to destination. At destination packets are re-assembled in right order. Router at each node requires storage to buffer packets

→ Virtual cut through routing

reduces storage needed and time for routing. This method buffers a packet only if next link to take is occupied. This reduces storage space & routing time

→ Wormhole routing

Packet is subdivided into flits (flow control digits). First flit has header info rest contain data All flits follow the same route as header flit. This routing is quite fast & needs much smaller buffer space. This may also be used in multistage interconnected networks.

PARALLEL ALGORITHMS

→ In order to simplify the design & analysis of parallel algo, parallel computers are represented by various abstract machine models.

MODELS OF COMPUTATION

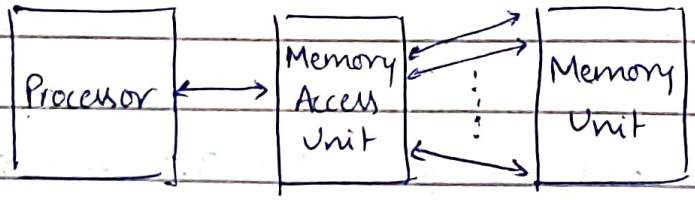
→ Various abstract machine models for parallel computers are:

- RAM
- PRAM
- Interconnection Networks
- Combinational Circuits

Random Access Machine (RAM)

→ This model abstracts sequential computer

→



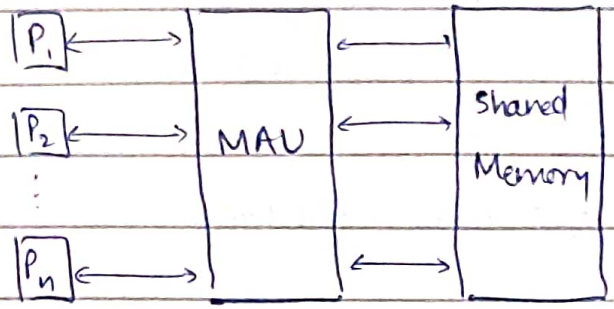
A memory ~~location~~ unit w/ M locations
A processor that operates under control of sequential algo.
A MAU which creates path from processor to arbitrary location in memory.

→ Any step of an algo for RAM model consist of 3 basic phases:

- Read → Processor reads data from memory
- Execute → Performs arithmetic / logical operation
- Write → Writes contents of its register to memory

Parallel Random Access Machine (PRAM)

→ Popular model for designing parallel algo.



- set of N identical processor
- Memory unit shared by all processors
- MAU which allows processors to access shared memory.

→ PRAM can be used to model SIMD & MIMD

→ When it models SIMD, all processors execute same algo synchron.

→ 3 phases, same as RAM, all O(1).

→ 4 categories:

Exclusive Read Exclusive Write (EREW)
Concurrent Read Exclusive Write (CREW)
Exclusive Read Concurrent Write (ERCW)
Concurrent Read Concurrent Write (CRCW)

→ Protocols to handle concurrent writes:

- Priority CW
- Common CW
- Arbitrary CW
- Combining CW

Interconnection Networks

→ Here instead of shared memory, processors communicate via direct links connecting them

→ M locations in memory distributed among N processors

Combinational Circuits

→ Refers to family of models of computation

→ can be viewed as a device having set of input lines at one end & set of output lines at other end. (arranged in columns)

→ made up of components, called stages.

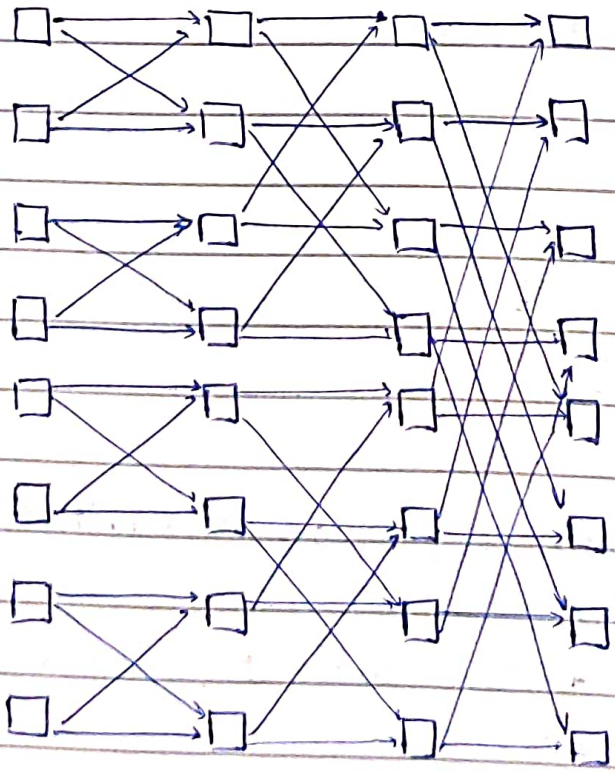
→ input lines = fan-in ; output lines = fan-out

→ 3 important parameters:

- size : no. of components
- depth : no. of stages i.e max no. of components on path from input to o/p.
- width : max components in a given stage.

→ **Butterfly circuit**

Used in designing a combinational circuit for efficient computation of FFT. It has n input, n output. Depth is $(1 + \log n)$ width is 'n'.



Analysis of Parallel Algorithms

→ 3 criteria
- Running Time
- No of processor
- Cost

Running Time

→ Defined as time taken by algo to solve problem on parallel comp.
→ is a function of I/p given to algo.
→ Parallel algo has 2 steps
  - computional : processor peforms operation
  - communication : data exchange b/w processors

→ Speedup

Defined as ratio of worst case running time of best sequential algo & worst case running time of parallel algo.

$$\text{speedup} = \frac{\text{running time of best seq. algo}}{\text{running time of } \| \text{ algo}}$$

Number of Processor

→ is a function of n. given by P(n).

Cost

→ Product of running time and no. of processors
→ In any step, some processors could be idle. Include them also.
→ If cost of parallel algo matches lower bound of best known sequential algo, then it is cost-optimal.
→ Efficiency (≤ 1)

$$= \frac{\text{Worst case running time of best sequential algo}}{\text{cost of parallel algo}}$$

PERFORMANCE METRICS

Following are the performance metrics:

- parallel run time
- Speedup
- efficiency

Parallel Run Time

→ Parallel run time, denoted by $T(n)$ is time required to run program on n-processor parallel computer

→ $T(1)$ denotes best sequential algorithm runtime

Speedup

→ Ratio of time taken on single processor to time taken on parallel comp w/ identical processors

$$S(n) = \frac{T(1)}{T(n)}$$

→ Theoretically, speedup can never exceed no. of processors, n.

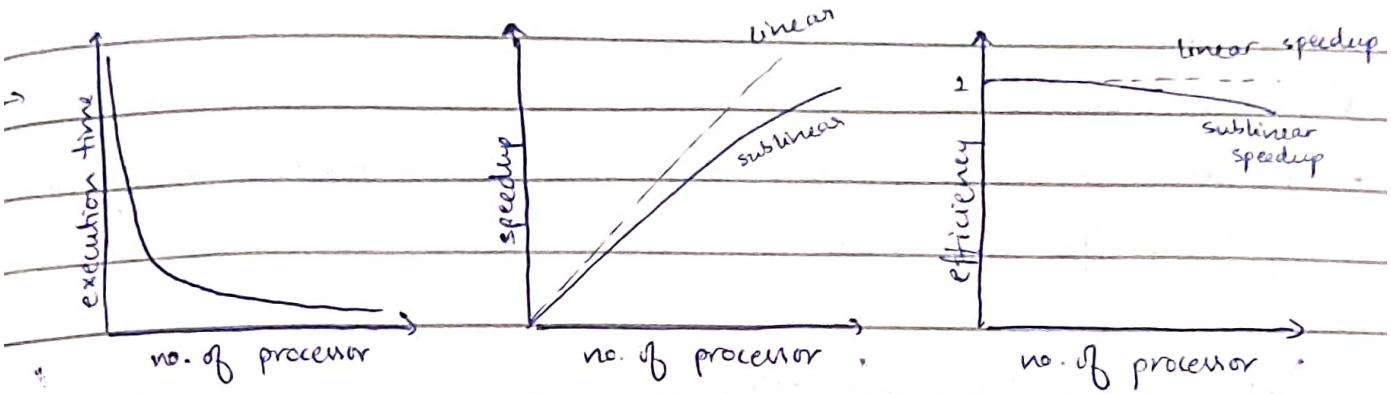
→ A speedup $> n$ possible if each processor spends less than $T(1)/n$ units of time for solving problem. This is usually due to:

- non-optimal sequential algo
- hardware characteristics
- speculative computation which put sequential algo at disadvantage

Efficiency

→ defined as ratio of speedup & no. of processors used to achieve it.

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n\,T(n)}$$



BENCHMARKS

→ Most computer manufacturers specify peak performance or sustained performance in terms of MIPS

→ Some benchmarks (a set of programs or prog. fragments) are used to compare performance of machines.

→ Small programs which are especially constructed for benchmarking purposes & do not represent any real computation are called synthetic benchmarks

Major Drawback: Don't reflect actual behaviour of real prog.

→ Program fragment which are extracted from real programs are called kernel benchmarks as they are heavily used core of programs

Major Drawback: Performance results they produce are very large.

→ Application benchmarks comprise several complete applications that reflect workload of a standard user

→ Performance results of benchmarks also depend on compilers used.

Parallel Overheads

→ Parallel computers don't achieve linear speedup or efficiency of 1 due to overheads

① Inter-process Communication

→ The time to transfer data b/w processors is usually the most significant source of parallel computing overhead.

→ If each processor spends t_{comm} time, the interprocess communication adds $(t_{comm} \times p \times \alpha_0)$ to parallel overhead

② Load-Imbalance

→ It is impossible to determine size of subproblems to be assigned to processors

→ If different processors have diff. workload, some may be idle while other work on problem.

③ Inter-task synchronization

→ Some or all processors must synchronize at certain points due to dependencies among its subtasks.

→ If all processors are not ready at same time, some processors will stay ideal which contributes to overhead.

④ Extra Computation

→ The best sequential algo may not have high degree of parallelism thus forcing to use parallel algo which has more operation count.

→ This is regarded as overhead cause well this is extra work!

⑤ Other Overheads

→ Like task creation, task scheduling, task termination, cache coherence enforcement etc.

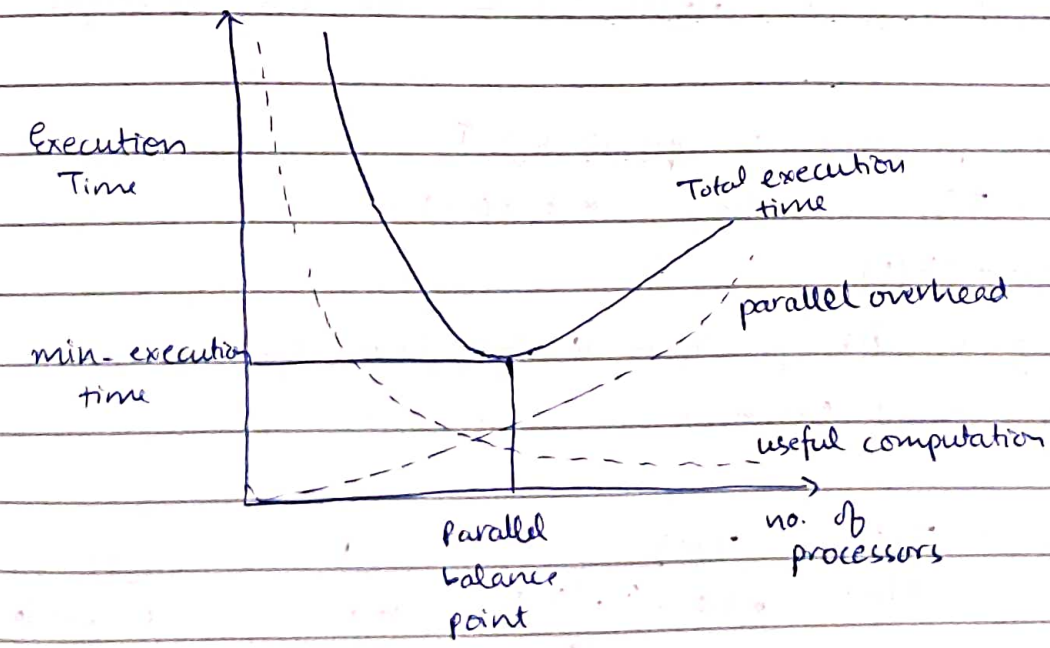
Balance Point

→ As no. of processors ↑, execution time ↓

→ At some point, processor's workload becomes comparable to its parallel overhead.

→ From this point, execution time ↑.

→ There is an optimal no. of processors to be used for a given problem. This is called parallel balance point, past which employing more processors w/out increasing problem size, increases execution time.



SPEEDUP PERFORMANCE LAWS

→ There are 3 speedup laws
→ Amdahl's law based on fixed problem size or workload
→ Gustafson's law is for scaled problems where problem size increases w/ increase in machine size
→ Sun and Ni's law is applied to scaled problems bounded by memory capacity.

Amdahl's law

→ In Amdahl's law, computational workload W is fixed while no. of processors that work on W can be increased.
→ According to amdahl, a prog contains 2 types of operations completely sequential which must be done sequentially and completely parallel which can be run on multiple processors
→ Let time for sequential operations = T_s

$T_s = \alpha . T(1)$ where $\alpha \in (0,1)$

$\therefore T_p = (1-\alpha) T(1)$

Assuming parallel operations achieve linear speedup i.e. ignoring parallel overhead

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{\alpha T(1) + \frac{(1-\alpha)T(1)}{n}} = \frac{1}{\alpha + \frac{1-\alpha}{n}} = \frac{n}{1+(n-1)\alpha}$$

→ Even if $n \to \infty$ $S(n) = 1/\alpha$. Thus speedup is limited. This is sequential bottleneck problem.
→ for multicore processors, $S(f,n) = \frac{1}{(1-f) + f/n}$

(f: portion that can be parallelized)

Gustafson's Law

→ This law says that increase of problem size for large machines can retain scalability w.r.t no. of processors.

→ As machine size increases, work load is ↑ so as to keep fixed execution time.

→ Let T_s = time for sequential operations

$T_p(n, W)$ = time for parallel operations on workload W w/ n processors

$$S'(n) = \frac{T_s + T_p(1, W)}{T_s + T_p(n, W)}$$

Like ~~Ahmd~~ Amdahl, we assume no parallel overhead,

$\Rightarrow T_p(1, W) = n \cdot T_p(n, W)$

Let α be fraction of seq. workload

$$\alpha = \frac{T_s}{T_s + T_p(n, W)}$$

$$\therefore S'(n) = \frac{T_s + n T_p(n, W)}{T_s + T_p(n, W)}$$

$$= \alpha + n(1-\alpha) = n - \alpha(n-1)$$

Sun and Ni's Law

→ Sun and Ni developed a memory-bounded speedup model which generalizes both amdahl & gustafson's law to maximise use of processor & memory capacity.

→ The idea is to solve maximum possible size of problem, limited by memory capacity.

→ For n nodes, assume parallel portion of workload is increased by G(n) times reflecting increase in memory

$$W^* = \alpha W + (1-\alpha) G(n) W$$

$$S^* = \frac{T_s + G(n) T_p(n,W)}{T_s + \frac{G(n)}{n} \cdot T_p(n,W)} = \frac{\alpha + G(n)(1-\alpha)}{\alpha + \frac{G(n)}{n}(1-\alpha)}$$

→ Depending on G(n) there are 3 cases:

Case 1 : $G(n) = 1$

$$S^* = \frac{\alpha + (1-\alpha)}{\alpha + (1-\alpha)/n} = \frac{1}{1+(n-1)\alpha}$$ (Amdahl's Law)

Case 2 : $G(n) = n$

$$S^* = \frac{\alpha + (1-\alpha)n}{\alpha + (1-\alpha)^n/n} = \alpha + (1-\alpha)n$$ (Gustafson's law)

Case 3 : $G(n) \geq n$

Here speedup is slightly greater than gustafson's law.

AKSHAT SARASWAT

Strong Scaling

→ A prog. is said to be strongly scalable if we increase no. of processor, we can keep efficiency fixed w/out increasing problem size.

→ Max speedup is $1/\alpha$

Weak Scaling

→ A prog. is said to be weakly scalable if we keep efficiency fixed by increasing problem size at same rate as we increase no. of processors.

XXX