UNIT-1 (contd.)

PARALLEL COMPUTING	DISTRIBUTED COMPUTING
* Many ops performed simultaneously	System components are located at different locations
* Single computer is required.	Uses multiple computers
* Multiple processors perform multiple ops	Multiple computers perform multiple ops
* May have shared or distributed memory	Has only distributed memory
* Processors communicate w/ each other through the bus.	Computers communicate w/ each other through message passing
* Improves system performance	Improves system scalibility, fault tolerance & resource-sharing capabilities

UNIT - 1

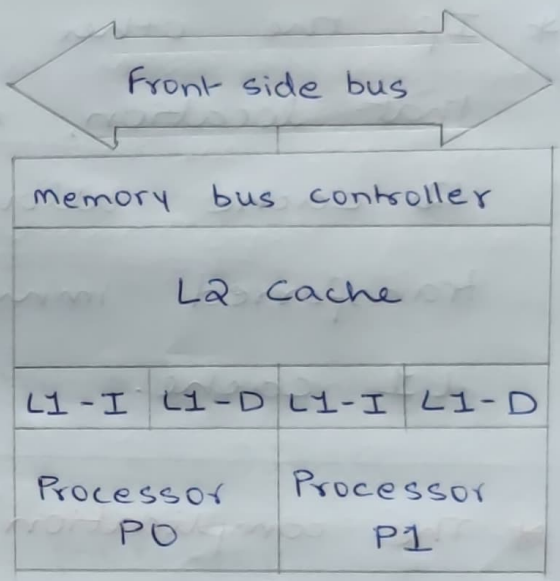
A LOOK AT 6 PARALLEL COMPUTERS

CHIP MULTIPROCESSORS

① Intel Core Duo

Design Features

* Two 32-bit Pentium processors on a single chip.
* 32K L1 data & instr. caches in each processor.
* Shared 2MB or 4MB L2 cache.
* Shared memory controller, I/O controller & so on.
* Fast on-chip comm. b/w the 2 processors through shared mem.



- The 2 processors have 32KB private level 1 caches for instr. & data
- These caches are supplied by the shared level 2 cache (Instr. & D)
- The bus controller mediates transfers b/w the L2 cache & the RAM via the FSB.

* Both processors see a consitent shared memory image.
* When a proc. references a location in RAM, the cache line containing that location is transfered to the L2 cache & then to the L1 cache of the requesting proc.
* If the other proc. also accesses that location, its presence in the L2 cache allows the line to be transferred immediatly to that proc.'s L1 cache.

* The complication comes when one of the proc. wants to change the value in a location.
* If 1 proc changes its private L1 copy, while the other continues to use its private L1 copy w/ the old value, then the computation would be incorrect. The old value is said to be stale

* Solution → Cache Coherency Protocol
* Ensures that a processor only writes to a privately cached loc. when it has exclusive use of that line.
* MESI protocol → Modified, Exclusive, Shared & Invalid

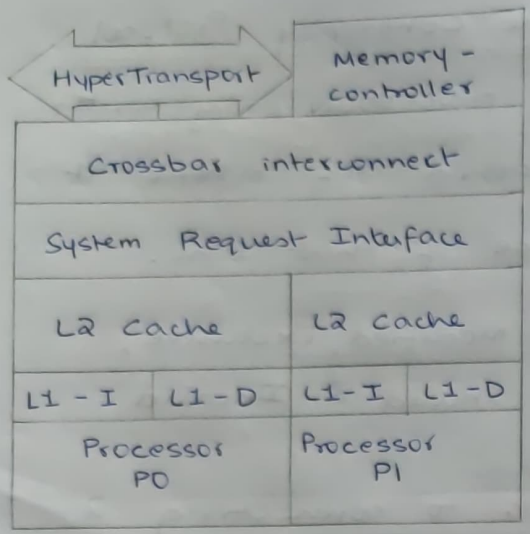
Other complications to connecting 2 cores on a chip:

* The protocol introduces overhead
* when the procs are working on a problem, then memory bandwidth requirements can approach twice the bandwidth required of a singe processor.

② AMD Dual Core Opteron

* Two AMD 64 procs on a single chip
* 64K L1 D & Inst. caches in each p.
* Seperate 1MB L2 cache per proc.

* Direct Connect Arch. for shared mem. access
* Fast on-chip comm. b/w the p's thru the System Request Interface.



- Requests to RAM or other p's are implemented using HyperTransport.

- Private L1 cache
- Private L2 cache
- SRI handles the mem. coherency responsibilities.
- MOESI protocol → MESI + "owned" state. Allows cache value to be shared among proc's even when the RAM copy is stale.

SYMMETRIC MICROPROCESSORS

① Sun Fire E25K

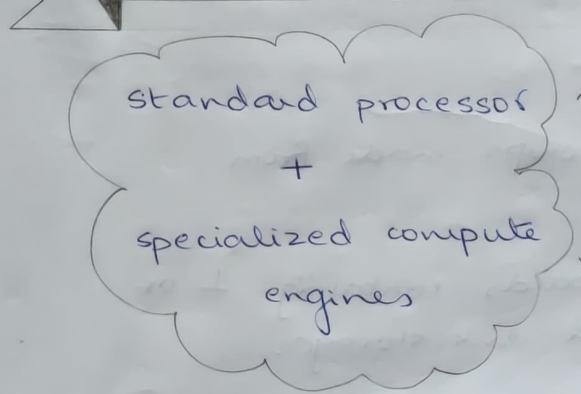
* Up to 72 processors, capable of executing 2 hardware threads.
* 150-MHz Sun Fireplane composed of 3 18x18 crossbar interconnects for addr., data & response, & 18 snoopy buses.
* Access latency to shared mem. is equal for all p's.
* Shared memory of 1.15 TB

* The architecture uses a directory-based cache coherency protocol among boards.
* In this, all mem. requests are sent to a centralized directory that maintains the state of all cached memory blocks.

* Longer latencies as each mem. operation consists of the orig. request to the directory, forwarding of the op. to the apt cache, & the subsequent response to the requesting processor.

* The crossbars provide a substantial amt. of comm. capability.

Heterogeneous Chip Designs



① Cell

* Joint dev → IBM, Toshiba, Sony
* Synergistic Procesing Elements (SPEs) - specialized cores supporting 32-bit vector op's.

* Dual threaded 64-bit PowerPC proc.
* 8 32-bit SPEs
* 256 KB on chip RAM for each SPE
* High speed EIB connecting the SPEs

(diagram - fb)

EIB - Element Interconnect Bus

* Doesn't provide coherent memory for the SPEs

(diagram - fb)

CLUSTERS

* They are parallel comp's made from commodity parts.
* The nodes are boards containing 1 or more proc.'s, RAM & disk storage.
* The nodes are connected by commodity interconnect
* Memory is not shared among machines; a procesor, which only accesses the mem. on its board, comm. w/ other p's by passing messages.

① HP Cluster Platform 6000 blade

* Any no. of blades, each w/ 2 dual core Itanium2 1.6 GHz p's w/ 3MB cache
* 16 GB RAM per blade
* Double disks per blade, Fiber Channel interconnect
* myricom myrinet 2000 interconnect

SUPERCOMPUTERS

① BlueGene/L - built by IBM

* 65,536 dual core nodes; each node is a 440 PowerPC proc.
* Each node has 32K L1 inst. & D. caches
* " " 4 ports to a barrier network
* Each node has Double Hummer optimized floating pt. units
* " " has a 4MB sequentially consistent shared on-chip L3 cache & 512 MB shared off chip RAM

* " " " 6 bidirectional ports to a 3-D torus network
* " " " 3 bidirectional ports to a collective network.

* Comm. b/w p's that aren't directly connected in the torus is routed along a path thru the network.

* The collective network is a 2nd, independent network connecting the nodes. The network chips have been augmented w/ arithmetic capabilities so that data flowing thru the network can be combined to form, say, global sums.

* Barrier network provides a 3rd type of global comm.

(diagram – tb)

THE RAM

(Random Access Machine)

* This model abstracts a sequential computer as a device w/ an instruction execution unit & an unbounded memory.

* The memory stores both program instr. & data, & any mem. location can be referenced in "unit" time w/out regard to its location

* The Instr. Exec. Unit fetches & executes 1 instruction every cycle, & unless directed otherwise by a branch, it proceeds to the next instr. in sequence on the next cycle.

* The model describes how a program will run, & for the model to be useful the hardware has to perform as described.

* This model is unrealistic.
* vector p's, which can fetch long vectors of data in a single cycle, don't fit the RAM model.

(contd. later)

THE PRAM

(Parallel RAM)

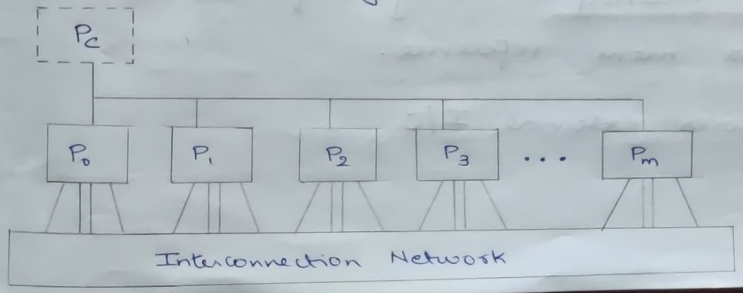
* Consists of an unspecified no. of instruction exec. units connected to a single unbounded shared mem. that contains both programs & data.
* The instr. exec. units can follow their own program threads, but they execute instr. in lock step, making synchronization easier.
* All exec. units reference the global memory & all observe a single sequence of memory state changes.

* One complication of this model occurs when multiple instr. access the same mem. location at the same time.
* This model doesn't work well as a model for programmers.
* It fails by misrepresenting mem. behaviour.
* The problem w/ the PRAM is that it is apparently impossible to realize the unit-time single mem. image for scalable machines.
* As the no. of exec. units increases, the delays required to keep the mem. image consistent grow dramatically.
* The model ignores communication cost.

SELF STUDY

THE CTA MODEL

* Consists of P std sequential computers, called processors or processor elements connected by an interconnection network (comm. network).
* The P + the 1st processor is the controller. Its purpose is to assist w/ ops like initialization, synchronization & so on.



* The topology of the CTA's interconn. network is not specified.

* A NIC (Network Interface Chip) mediates the processor/network connection.

* Node degree → No. of wires connecting the p's to the network.
  It is a property of the topology.

* Data is read/written from/to memory by DMA.

* The p's execute independently, running their own local programs.

* Data references can be made to a p's own local memory or non-local memory.

  - shared-mem.
  - 1-sided comm
  - msg passing

* Memory latency - Delay required to make a mem. reference.

* CTA ignores external I/O

* Local mem. access time is the usual mem. access time of the sequential proc.

* Non-local mem. access time can be b/w 2-5 orders of magnitude larger than local MAT.

RAM Model (contd.)

Applying the RAM Model

* The simplicity of this model is essential as it allows programmers to estimate overall performance based on instr. counts w/in the model.

Eg: If we want to find an item that might be in an array A of sorted items, we could use → sequential search / binary search

→ Given the RAM model, we know that the S. search will take an avg. of n/2 iterations of the for loop to find the item, each iteration will require executing fewer than a dozen machine instructions.

→ Bin. search is more complex, but its expected performance is approx. $\log_2 n$ iterations of the while loop → fewer than 2 dozen machine instructions.

→ For small values of n, seq. search will be faster; bin. search will be more efficient for large values of n.

than

MEMORY REFERENCE MECHANISMS

① Shared Memory

* Difficult to program
* Encourages the creation of inefficient programs by making it too easy to make non-local references.
* Presents a single coherent image to multiple threads & requires some degree of hardware support to make it perform well.
* The risk of shared mem. is that 2 or more threads will attempt to change the same loc. at the same time.

② 1-sided comm.

* Supports a single shared address space, but doesn't attempt to keep the memory coherent.
* This (↑) simplifies the hardware but places great burdens on the programmer.
* Here, all addresses except those explicity designated as private can be referenced by all p's.
* get() , put()

③ Message Passing

* Requires the least hardware support
* Proc. can only access local memory
* To reference non-local data, msgs are used.
* send() , recv()
* Source & dest. processors must collaborate to transfer data

UNIT-2

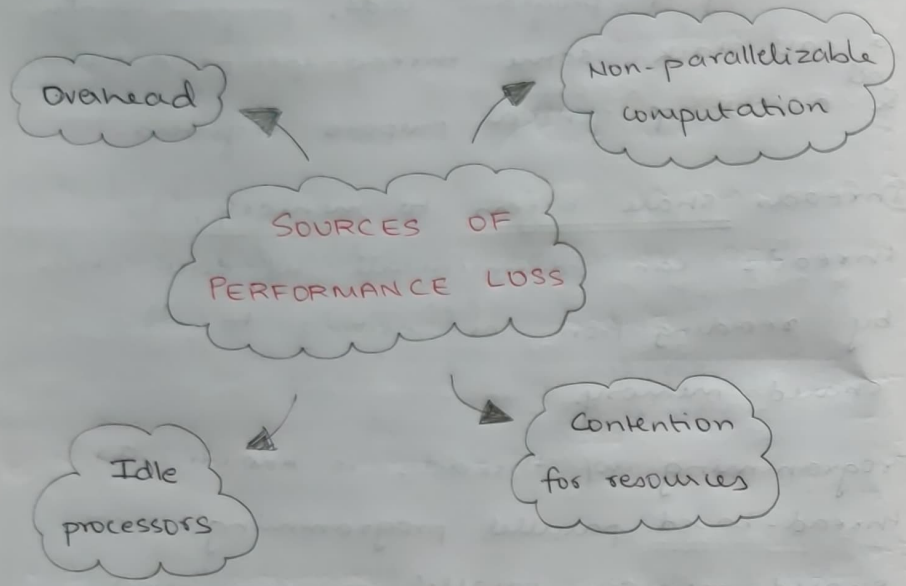
Threads

* Refers to a thread of control, logically consisting of program code, program counter, a call stack, some thread-specific data incl. a set of general purpose registers.
* Threads share access to memory, so threads can communicate w/ each other by reading from or writing to the shared memory.
* Programming w/ threads is known as thread-based parallel programming or shared-memory parallel programming

Processes

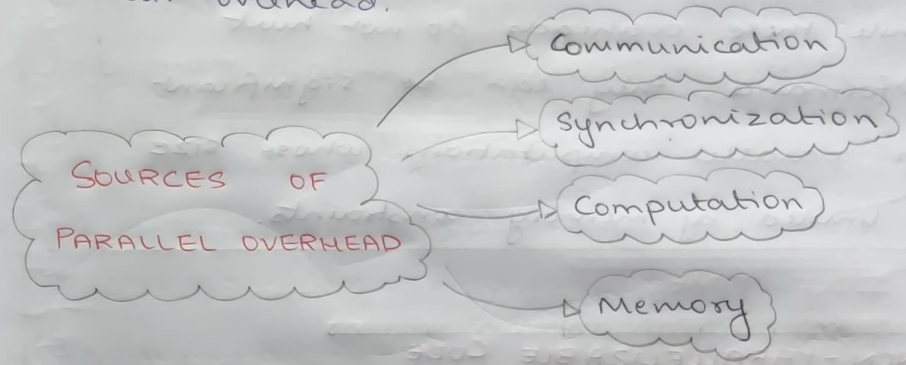
* It is a thread that also has its own private address space.
* Processes communicate w/ one another by passing msgs.
* Parallel programming w/ processes → message passing P.P or non-shared memory P.P

* Latency - Amt. of time it takes to complete a given unit of work.
* Throughput - Amt. of work that can be completed per unit time



① OVERHEAD

* Any cost that is incured in the parallel soln. but not in the serial soln is considered overhead.
* There is overhead in setting up threads & processes to execute concurrently & also for tearing them down.
* Memory allocation & initialization are expensive, ∴ processes incur greater setup overhead than threads.
* After the 1st process is set up, all subsequent threads & process setups incur overhead.



Comm.

* Since a sequential computation doesn't have to communicate w/ another processor, all comm. here is a form of overhead.

Synch.

* Overhead that arises when 1 process/thread must wait for an event on another thread/process
* It is implicit in many forms of msg passing, but often explicit when programming w/ threads.

Comp.

* Parallel computations almost always perform extra computations that aren't needed in the sequential soln.

Mem.

* While these overheads do not hurt performance, they can be significant for parallel computations whose size is limited by memory constraints.

② NON-PARALLELIZABLE CODE

* If a computation is inherently sequential (cannot be parallelized), then using more processors won't improve its performance.
* Sometimes this code is expresed as redundant code. (like initialization)

** AMDAHL'S LAW

* The existence of non-parallelizable computations limits the potential benefit from parallelization.
* This law observes that if $1/s$ of a computation is inherently sequential, then the max. performance improvement is limited to a factor of s
* The reasoning is that the execution time (T_p) of a parallel computation will be the sum of the time for its sequential component & its parallelizable component.

  $T_s \rightarrow$ time for sequential computation.

$$T_p = \frac{1}{s} \cdot T_s + \left(1 - \frac{1}{s}\right) \cdot \frac{T_s}{P}$$

  $P \rightarrow$ No. of processors

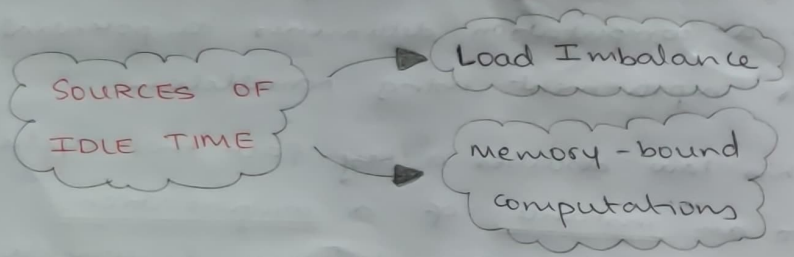
* Proportion of sequentially executed code in a computation determines its potential for improvement using parallelism.
* This law describes a key fact that applies to an instance of a computation.
* The law fixes an instance & considers the effect of increasing parallelism.

③ CONTENTION

* Degradation of system performance caused by competition for a shared resource.
* Its effects can often lead to shutdown.
* Eg: Lock contention creates excessive load on the memory.

④ IDLE TIME

* A process or thread might not be able to proceed due to a lack of work or because it is waiting for some external event.
* It is often a consequence of synchronization & communication.

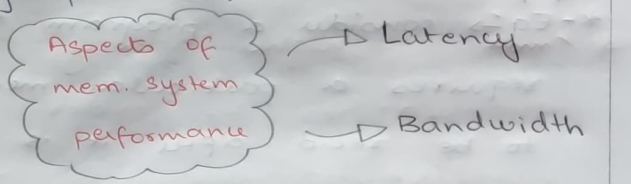


Load Imbal.

* Uneven distribution of work to processors.
* Extreme eg: when A sequential computation runs on 1 processor of a parallel machine, while the other processes remain idle.

Mem.-Bound Comp.

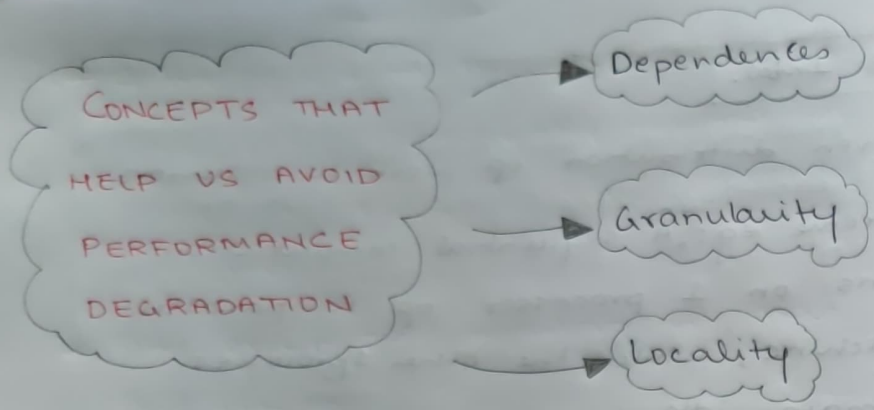
* A processor may stall if it is waiting for a memory operation, such as a read from DRAM.



* DRAM latencies have improved at a slower rate than processor speeds, so the distance to DRAM in terms of CPU cycles continues to increase.
* When data can be kept in caches, these latencies can be avoided.

Imp. of DRAM bandwidth:

* Latency-hiding techniques (like prefetching) introduce extra memory traffic
* Even computations that can reside in cache must consume mem. bandwidth to load caches.



① DEPENDENCES

* It is an ordering relationship b/w 2 computations.

* Eg:- consider a program that requires a particular mem. location to be read after it is written.

For the correct result, there is a dependence b/w the write & read ops.

* If the order of the 2 ops is skewed in time so that the read happens before the write, the dependence would be violated

DATA DEPENDENCE

* It is an ordering on a pair of mem. ops that must be preserved to maintain correctness.

(TYPES)
- Flow ·D — read after write → true depend
- Anti ·D — write after read → false "
- Output ·D — write after write → false "
- input — read after read

- true dependences → represent fundamental orderings of mem. ops
- false " → Arise from the reuse of mem.

** when creating a parallel program, we must be careful to avoid introducing dependences that don't matter to the computation, as they will unnecessarily limit parallelism.

② GRANULARITY

* The granularity of parallelism is determined by the frequency of interactions among threads or process.

* Frequency is measured in the no. of instructions b/w interactions.

* Coarse grain → Threads / processes that only infrequently depend on data or events in other threads/ processes.

* Fine grain → computations that interact frequently.

* Hardware platforms that implement low-latency communication (like multicore chips), support finer-grained comp's

* " " " higher-latency ", larger granularities are better, cuz the overhead of interaction is higher.

③ LOCALITY

Computations can exhibit both:

- Temporal locality – mem. references that are clustered in time
- Spatial " – " " " by address.

* It has the added benefit of minimizing dependences, thereby reducing overhead & contention.

MEASURING PERFORMANCE

① Execution Time

* Also called latency, it is the time elapsed from the point at which the 1st processor begins executing the program to the time the last processor completes execution.

* FLOPS (Floating pt. ops per sec) - another important metric.
Downside of using FLOPS - ignores other costs such as integer computations.

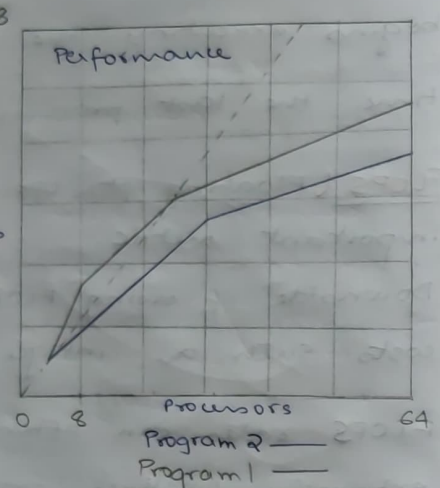
* FLOPS rates can often be affected by extremely low-level program modifications that allow the programs to exploit a special feature of the hardware.

* Limitation of both these metrics is that they distill all performance into a single number w/out giving any indication of the parallel behaviour of the computation.

② Speedup

* $\text{Speedup} = \frac{T_s}{T_p}$, T_s -> sequential time, T_p -> Parallel time

* It is usually plotted on the y-axis w/ the no. of processors on the x-axis.



↳ linear speedup

* This graph shows a feature typical of many parallel programs → the speedup curves level off as we increase the no. of processors.

* This is the result of keeping the problem size const. while increasing the no. of processors.

* ↑ this causes the work per processor to decrease & the overhead to increase, causing total execution time ~~to~~ not to improve linearly.

③ Superlinear Speedup

* A phenomenon sometimes occurs in which a parallel program runs more than a factor of P times faster than its sequential counterpart, yielding superlinear speedup.

* Explanation → the parallel program does less work.

* Eg: performing a search which is terminated as soon as the desired element is found.
  Sequential searches thru more data.
  Parallel is more efficient (less work).

④ Efficiency

* Normalized measure of speedup that indicates how effectively each processor is used. $\text{Efficiency} = \frac{\text{Speedup}}{P}$

* It is typically < 1 & diminishes as the no. of Processors is increased.

THE PERIL-L NOTATION

A programming lang. that can be used to develop & analyze parallel algos.

Goals of this lang:

- Should be minimal so that it's easy to learn.
- " " universal so that it doesn't bias toward any 1 lang., parallel comp., or algorithmic approach.
- " allow us to reason about performance.

Example (forall statement)

```
forall ( index in (1..12))
{
   printf (" Hello, from thread %i \n", index);
}
```

// This code ↑ prints a dozen times, in some unknown order. //

Exclusive block → mutual exclusion

```
forall ( index in (1..12))
{
   exclusive
   {
      printf (" Hello..%i", index);
   }
}
```

Barrier Synchronization → forces threads to stop & wait until all threads have arrived at the barrier, then they can proceed.

```
forall ( index in (1..12))
{
   printf (" tweedle dee\n");
   barrier;
   printf (" tweedle dum\n");
}
```

Variable declared outside a forall stmt → global

" " w/in " " → local copy for each thread.

Declaring FE variables → int t' = 0;

Reduce & Scan → +/count → Reduce & add elements of count

min\items → Scan, find the smallest of items

Count 3s computation (Try 3)

```
int array[length];
int t;                          No. of threads
int total = 0;                  Grand total
int lengthPer = ceil(length / t);   dividing work
forall (index in (0..t-1))
{
    int priv_count = 0;
    int i, myBase = index* lengthPer;
    for (i = myBase; i< min (myBase +lengthPer,
                             length); i++)
    {
        if (array[i] == 3)
        {
            priv_count++;
        }
    }
    exclusive { total += priv_count; }
}
```