UNIT-2. DC

(1) State and explain Amdhal's law for measuring speed up perfor. with outcomes.

* Amdhal's law observes that if 1/S of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of S.
* Execution time T_p of a parallel computation is sum of time for its sequential component and its parallelizable component.
* If computation takes T_s time to execute sequentially, then for P processors we have

$$T_p = 1/s * T_s + (1-1/s) * T_s/p$$

* The proportion of sequentially executed code in a computation determines its potential for improvement using parallelism.

* Issues / Problems

1. parallel implementation often executes more total instructions than the sequential solution, making the $(1-1/s) * T_s$ an underestimate
2. parallelizable portion of the computation might not be improved to maximum extent as there might be an upper limit on the no. of processors that can be usefully employed.

* Portions of a computation that are sequential will dominate the execution time
* Amdhal's law describes a key fact that applies to an instance of a computation
* The proportion of sequential code diminishes as larger instances are considered

(2) 26-way parallel solution to alphabetizing [Peril-L]

```
rec L[n];
forall (j in (0..25))
{ int myAllo = mySize(L,0);
  rec LocL[] = localize(L[]);
  int counts[26] = 0;
  int i, j, startPt, myLet;
  for (i=0; i<myAllo; i++
      counts[letRank(charAt(LocL[i].x, 0))] ++;

  counts[index] += +/counts[index];
  myLet = counts[index];
  rec Temp[myLet];
  j=0;
  for (i=0; i<n; i++)
    if (index == letRank(charAt(L[i].x, 0)))
        Temp[j++] = L[i];

  alphabetizeInPlace(Temp[]);
  startPt = +\myLet;
  for (i=0; i<count; i++)
      L[j++] = Temp[i];

}
```

Fixed Pallelism solution to Count 3s [Peril-L]

```
int array[length]; total;
int seg = ceil(length/4);
forall(j in (0..3))
{ int priv_count = 0;
  for (i = 0+seg; i < min(length, j+(seg+1)); i++)
      if (array[i] == 3)
          priv_count++;

  total = +/priv_count;
}.
```

(4) Notes on:

(1) Dependences:

* A dependence is an ordering relationship b/w 2 computations.
* Eg: a dependence can occur between 2 processes when one process waits for a message to arrive from another process.
* Dependences can also be defined in terms of read & write operations, i.e., memory loads & stores for threaded computation.
* Dependences ~~allow~~ provide a general way to describe limits to parallelism. They are not only useful for reasoning about correctness but also provide a way to reason about potential sources of performance loss.

* Data Dependences:
1. Flow depence – read after write – called true dependence
2. Anti dependence – write after read – called false dependence
3. Output dependence – write after write – called false dependence.

② Granularity:

* Granularity can be typically described by using the term like coarse and fine.
* Granularity of parallelism - can be determined by the freq. of interactions among threads/processes, i.e., frequency with which dependences cross thread/process boundaries.
* It is important to match the granularity of the computation with hardware's available resources and the solutions particular needs.
* Coarsest computation involve huge amounts of computation and almost no interaction
* Coarse grain - infrequent depend on data/events in other thread/proc.
  Fine grain - computations that interact frequently with other threads.
* Hardware platforms that implement low-latency communications support finer-grained computations.
* Message passing programs work best with coarse grained computations.
* Fine grained threads with several instructions b/w interactions, implement low-latency communication among processors residing on the same chip.

③ Scalability & Performance Issues:

* Scalable Performance is Difficult to achieve -
  as the no. of processors increase, it becomes difficult to achieve good parallel efficiency.
  ↳ Overhead must be minimized as the parallel system expands

* Implications to Hardware -
  As the no. of processors increase, the marginal benefit of improving each processors' CPU speed is minimal.

↳ Amdhal's law suggests that better speedup and efficiency can be attained by using slower cores.
↳ Eg: BlueGene/L – processors = 64k has relatively slower processors compared to other machines.

* Implications for Software:
The notion is that larger problem sizes yield better parallel performance is implicit in the half-performance metric ($n_{1/2}$). which measures for a given prog. & machine, the input size needed to obtain an efficiency of one-half.

* Scaling the Problem Size:
Basic algorithm needs to be as scalable as possible, & simply adding more processors doesn't change the fact – it exacerbates it. This argument is called corollary of modest potential

(5) What are the Sources of Performance Loss?

→ Sources of performance loss are as follows:
1. Overhead – any cost incurred in parallel computation but not in serial solution is considered to be an overhead.
↳ Communication – communication among threads and processes is a major component of overhead. The specific costs of communication will depend on the details of the hardware.
↳ Synchronization – synchronization is a form of overhead that arises when one thread/process has to wait for an event to occur on another thread/process. Synchronization is implicit in many forms of message passing, while it is often explicit when programming with threads.
↳ Computation – parallel computation always perform extra computation which is not required in case of a serial solution.
↳ Memory – often parallel computation requires extra memory than serial sol^n.

2. Non-Parallelizable Computation – Amdhal's Law

* if a computation is inherently sequential; using more processors will not improve its performance.
* explain Amdhal's law

3. Contention –

* degradation of system performance caused by competition for a shared resource

4. Idle Time –

* Load Imbalance – uneven distribution of work to processors.
* Memory Bound Computation – when waiting for a memory operation such as a read from DRAM.

(6) Fixed, Unlimited, Scalable Parallelism.

(7) Performance measuring parameters.

Performance Trade Offs :

1. Communication vs. Computation

Comm^n costs are often significant source of overhead. It is possible to reduce comm^n by performing additional computation.

(1) Overlapping Comm^n and Computation –
The key is to identify computation that is independent of the comm^n, by executing computation & comm^n concurrently, latency of comm^n can be partially hidden.

(2) Redundant Computation –
recompute a value locally rather than wait for it to be transmitted

2. Memory vs. Parallelism

Parallelism can often be increased at the expense of increased memory usage.

(1) Privatization –
parallelism can be increased by using additional memory to break false dependences.

(2) Padding –
parallelism can also be increased by padding, allocating extra memory to force variable to reside on their own cache line

3. Overhead vs Parallelism

As we increase the no. of threads, the parallelism likely increases but so does overhead & contention.

(1) Parallelize Overhead –
threads divide the task of accumulating intermediate values into several independent parallel activities, thereby parallelizing some overhead and reducing its cost.

4 (2) Load balance vs. Overhead -

increased parallelism can also improve load balance, as it is often easier to distribute evenly a large no. of fine grained units of work.

The disadv. of this is an increase in overhead and in comm & synchronization

5. Granularity Trade Offs

One way to reduce the no. of dependences is to increase the granularity of interaction.

Batching is a programming technique in which work is performed as a group.

* Performance Measure Parameters.

1. Execution Time -

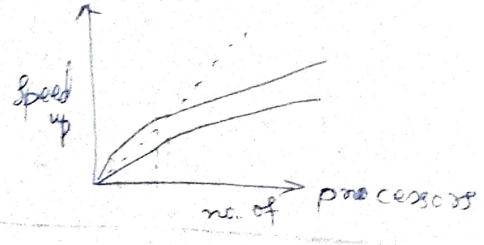
a) Latency - is the time elapsed from the point at which the first processor begins executing the program to the time the last processor completes execution.

b) FLOPS - floating-point operations per second.
It ignores other costs such as integer computations which may be significant component of computation time

→ Limitation of both metrics is they distill all performance into a single number without giving any indication of parallel behaviour of the computation.

2. Speedup -
defined as the execution time of a sequential program divided by the executiontime of a parallel program that computes the same results.

$$\text{Speedup} = T_s / T_p$$



Super Linear Speedup -

when a parallel program runs more than a factor of P times faster than its sequential counterpart.

4. Efficiency -

is a normalized measure of speedup that indicates how effectively each processor is used.

$$\text{efficiency} = \frac{\text{speedup}}{P}$$

efficiency > 1 ⟶ superlinear speedup case.

* Parallelism Trough PERIL - L

→ Peril-L begins as a sequential thread.
→ Multiple threads are introduced by the forall statement

```
forall (<integer variable> in (<index range>))
{   <body>
}
```

→ Eg:
```
forall (index in (1..4))
{ printf ("Hellow, thred %i \n", index);
}
```

* Goals of Peril-L

1. shoulld be minimal so that it is easy to learn.
2. should allow users to reason about performance
3. should be universal so that it does not bias towards any one language, parallel computer, or algo approach

* The Peril-L Notation

Peril-L is a programming language that can be used to develop and analyze parallel algos.

* Data Parallelism vs Task Parallelism

DIFF. FORMS OF PARALLELISM

→ Data parallelism is applied by performing the same opr. to different items of data at the same time.

→ Task parallelism is applied by performing distinct computation - or tasks - at the same time. Since the no. of tasks is fixed, the parallelism is not scalable

→ the amount of parallelism grows with the size of data.

→ Bit-level parallelism.
→ Instruction-level parallelism.

* Connecting Global & Local Memory

LOCALIZATION OF GLOBAL MEMORY

→ global data structures are distributed throughout the local memories of the processors.
→ global addressing is implemented by translating global addresses into appropriate local addresses of specific processors.
→ the localize() function refers the globally named data with a local name. & is used to initialize the local data structure.

```
int allData[n];
forall (threadID in (0 ... P-1))
{ int size = n/P;
  int locData [size] = localize (allData[]);
  ...
}
```

→ The localize function returns a reference to the portion of the global data structure that is allocated to the processor on which the thread executes.

→ all modifications of local data structure using local name are equivalent to modifications of the global data structure but without the λ-penalty.

→ issues with localization:

1. there is no local copy — global & local references are to the same memory location.
2. if multiple threads are assigned to a processor, each thread operates on only its part of the global data structure
3. Arrays containing localized data use local indices, so the first item of the local array is index 0.