Instructor Guide

In parallel processing, rather than having a single program execute tasks in a sequence, parts of the program are instead split such that the program is executed concurrently (i.e. at the same time), by multiple entities.

The entities that execute the program can be called either threads or processes depending on how memory is mapped to them.

One way of categorizing parallel computers is by the approach they employ in handling instructions and data

Flynn's taxonomy- still the most common way of classifying systems with parallel processing capability in the following categories of computer systems:

Single instruction, single data (SISD)
Single instruction, multiple data (SIMD)
Multiple instruction, single data (MISD
Multiple instruction, multiple data (MIMD)

Single instruction, Single data (SISD) : These are the classic (von Neumann) serial computers executing a single instruction on a single data stream before the next instruction and next data stream are encountered . A single processor executes a single instruction stream to operate on data stored in a single memory. An example of this category machine is Uniprocessors.

Single instruction, Multiple data (SIMD) : Here instructions are processed from a single stream, but the instructions act concurrently on multiple data elements. Generally the nodes are simple and relatively slow but are large in number. Thus, simultaneous execution of a number of processing elements is controlled by a

single machine instruction on a lockstep basis. In SIMD, each processors has data memory and different processors execute instructions on different sets of data. Some examples of this category machine are Vector and array processors.

Multiple instruction, Single data (MISD) : Here different instruction sequence is executed on same data stored on a set of processors,. Applications for this category machines are much less common than other categories.

Multiple instruction, Multiple data (MIMD) :
In this category each processor runs independently of the others with independent instructions and data, and a different instruction sequences on different data sets are executed simultaneously on a set of processors
These are the types of machines that employ message–passing packages, such as MPI, to communicate among processors.
They may be a collection of workstations linked via a network, or more integrated machines with thousands of processors on internal boards. These computers, which do not have a shared memory space, are also called multicomputers.
Although these types of computers are some of the most difficult to program, their low cost and effectiveness for certain classes of problems have led to their being the dominant type of parallel computer at present.
Some examples of this category machine are symmetric multiprocessor (SMP), clusters, and non-uniform memory access (NUMA) systems fit into this category

There are three possible parallel computers' memory architectures:

1. Shared memory
Uniform Memory Access (UMA)
Non-Uniform Memory Access (NUMA)

2. Distributed memory

3. Hybrid Distributed Shared memory

Shared memory

In shared memory parallelism, threads share a memory space among them. Threads are able to read and write to and from the memory of other threads.
One of the standard for shared memory considered is OpenMP, which uses a series of pragmas, or directives for specifying parallel regions of code in C, C++ or Fortran to be executed by threads.
In this architecture, the programmer's task is to specify the activities of a set of processes that communicate by reading and writing shared memory.
Uniform Memory Access (UMA): in this architecture, the identical processors have equal access times to memory; commonly used in symmetric multiprocessor (SMP) systems.
Non-Uniform Memory Access (NUMA):): in this architecture many SMPs are linked together, however all processors donot have equal access times to the memories of allother SMPs. Moreover, the memory access across the link is slower.

Distributed memory

In contrast to shared memory parallelism, in distributed memory parallelism, processes each keep their own private memories, separate from the memories of other processes.
In order for one process to access data from the memory of another process, the data must be communicated, commonly by a technique known as message passing, in which the data is packaged up and sent over a network.
In this architecture, the programmers have explicit control over data distribution and communication. Synchronization between tasks is programmers responsibility.
One standard of message passing is the Message Passing Interface (MPI), which defines a set of functions that can be used inside of C, C++ or Fortran codes for passing messages.

Hybrid memory

A third type of parallelism, in which both shared and distributed memory are utilized. The largest and fastest computers in modern world employ both shared and distributed memory architectures.

Shared component in this architecture is SMP machine and distributed component here is the network of multiple SMP's.
In hybrid parallelism, the problem is broken into tasks that each process executes in parallel.
The tasks are then broken further into subtasks that each of the threads execute in parallel.
After the threads have executed their sub-tasks, the processes use the shared memory to gather the results from the threads.
The processes use message passing to gather the results from other processes.

Speedup
is a notion that a program will run faster if it is parallelized as opposed to executed serially.

Accuracy
is the notion of forming a better model of a problem.

Weak Scaling
Is a notion that more processes and threads can be used to solve a bigger problem in the same amount of time it would take fewer processes and threads to solve a smaller problem

There are many issues that limit the advantages of parallelism.
Some of them are :
Communication Overhead
Amdahl's Law

Communication Overhead
refers to the time that is lost waiting for communications to take place in between calculations.
During this time, valuable data is being communicated, but no progress is being made on executing the algorithm.

The communication overhead of a program can quickly overwhelm the total time spent solving the problem, sometimes even to the point of making the program less efficient than its serial counterpart.
Communication overhead can thus mitigate the advantages of parallelism.

Amdahl's law:
A second issue is described in an observation put forth by Gene Amdahl and is commonly referred to as Amdahl's Law.
Used to characterize the performance of parallel algorithms.
Amdahl's Law says that the speedup of a parallel program will be limited by its serial regions, or the parts of the algorithm that cannot be executed in parallel.
Amdahl's Law posits that as the number of processors devoted to the problem increases, the advantages of parallelism diminish as the serial regions become the only part of the code that take significant time to execute. In other words, a parallel program can only execute as fast as its serial regions.

Amdahl's Law is represented as an equation below:

$$\text{Speedup} = \frac{1}{(1-P)+\frac{P}{N}}, \text{ where}$$

P = the time it takes to execute the parallel regions
1 – P = the time it takes to execute the serial regions
N = the number of processors

The theoretical maximum speedup of a program as a function of the fraction of the program that potentially may be run in parallel. The different curves correspond to different numbers of processors.

Amdahl's Law shows us that a program will have diminishing returns in terms of speedup as the number of processors is increased.
However, it does not place a limit on the weak scaling that can be achieved by the program, as the program may allow for bigger classes of problems to be solved as more processors become available.
The advantages of parallelism for weak scaling are summarized by John Gustafson in Gustafson's Law.

Gustafson's Law
states that bigger problems can be solved in the same amount of time as smaller problems if the processor count is increased. Gustafson's Law is represented as an equation below.

$$\text{Speedup}(N) = N - (1 - P) * (N - 1)$$

where

N = the number of processors

(1 – P) = the time it takes to execute the serial regions

Amdahl's Law reveals the limitations of what is known as strong scaling, in which the problem size remains constant as the number of processors is increased.
This is opposed to weak scaling, in which the problem size per processor remains constant as the number of processors increases, but the overall problem size increases as more processors are added.