Parallel computing

- To be run using multiple CPUs
To be run using multiple CPUs
A problem is broken into discrete parts that can be solved concurrently
Parallel computing

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
Parallel computing

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs
A single computer with multiple processors;
Parallel computing models

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
Parallel computing models

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.
The *wall-clock time* spent on the computation is the total time between the beginning and end of the computation process.
• The *wall-clock time* spent on the computation is the total time between the beginning and end of the computation process.

• The actual computing time, is the sum of the times spent by each of the processors.
The *wall-clock time* spent on the computation is the total time between the beginning and end of the computation process.

The actual computing time, is the sum of the times spent by each of the processors.

The speedup due to parallelising code is defined as

\[
\text{speedup} = \frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}
\]
The *wall-clock time* spent on the computation is the total time between the beginning and end of the computation process.

The actual computing time, is the sum of the times spent by each of the processors.

The speedup due to parallelising code is defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

The main reason for parallelising code is to save on wall-clock time using the availability of parallel programming environments.
There is always an overhead in setting up the parallelism.
There is always an overhead in setting up the parallelism. The effort is worth it only if the total gain is significant after accounting for parallelising overhead.
• There is always an overhead in setting up the parallelism.
• The effort is worth it only if the total gain is significant after accounting for parallelising overhead.
• Thus it is better for large execution instances.
There is always an overhead in setting up the parallelism.

The effort is worth it only if the total gain is significant after accounting for parallelising overhead.

Thus it is better for large execution instances.

The fraction of a code which is inherently serial limits the extent of parallelisability, regardless of the number of processors available.
There is always an overhead in setting up the parallelism.

The effort is worth it only if the total gain is significant after accounting for parallelising overhead.

Thus it is better for large execution instances.

The fraction of a code which is inherently serial limits the extend of parallelisability, regardless of the number of processors available.

If $S$ is the fraction of serial code, then, ignoring overhead, if there are $N$ processors, the speedup is $\frac{1}{S} + \frac{1 - S}{N}$.
There is a four way classification, based on Data and Code
There is a four way classification, based on Data and Code

**SISD** stands for Single-instruction single data.
There is a four way classification, based on Data and Code.

- **SISD** stands for Single-instruction single data.
- The other three are all combinations of single and multiple instructions and data.
There is a four way classification, based on Data and Code.

SISD stands for Single-instruction single data.

The other three are all combinations of single and multiple instructions and data.

The first category is merely a serial computer.
Domain decomposition consists of breaking the data set on which the problem operates into smaller parts which can be manipulated in parallel (simultaneously).
- **Domain decomposition** consists of breaking the data set on which the problem operates into smaller parts which can be manipulated in parallel (simultaneously).

- **Task decomposition** decomposes the work to be performed into a number of distinct parts, and each part is performed in parallel on the entire data set.
**Domain decomposition** consists of breaking the data set on which the problem operates into smaller parts which can be manipulated in parallel (simultaneously).

**Task decomposition** decomposes the work to be performed into a number of distinct parts, and each part is performed in parallel on the entire data set.

**Pipelining** is the process of splitting code to be run repeatedly over a long stream of input instances into logical units, which must execute in a specific order. Specifically, the output of each element in the pipeline is passed as input to the next element. This is like an assembly line.
Other models

- Shared Memory
Other models

- Shared Memory
- Threads
Other models

- Shared Memory
- Threads
- Message Passing
Memory architectures

- We have uniform memory access within a computer.
We have uniform memory access within a computer.

There is also, non-uniform memory access. Both these are under shared memory.
Memory architectures

- We have uniform memory access within a computer.
- There is also, non-uniform memory access. Both these are under shared memory.
- There is a distributed memory, with memory local to each computer.
Manual parallelisation requires the programmer to figure out where there is scope for parallelism in the program and implement it accordingly.
Manual vs. Automatic Parallelism

- Manual parallelisation requires the programmer to figure out where there is scope for parallelism in the program and implement it accordingly.

- Automatic parallelisation is figured out by compilers running serial code, by looking for inhibitors to parallelism.
Manual parallelisation requires the programmer to figure out where there is scope for parallelism in the program and implement it accordingly.

Automatic parallelisation is figured out by compilers running serial code, by looking for inhibitors to parallelism.

There is a combination of automatic, with programmer intervention.