Multicore Processor
A single computing component with two or more independent cores
Multiple cores run multiple instructions at the same time (concurrently)
Increase overall program speed (performance)

Manycore processor (GPU)
multi-core architectures with an especially high number of cores

Process
run in separate memory address space
a program in execution

Thread
Run in shared memory space in a process
One process may have multiple threads

Multithreaded Program
A program running with multiple threads that is executed simultaneously

Parallel Computing
Using multiple processors in parallel to solve problems more quickly than with a single processor
All processors may have access to a shared memory to exchange information between processors
More tightly coupled to multi-threading

Parallel Programming
Using additional computational resources to produce an answer faster

Concurrent Programming
Correctly and efficiently controlling access by multiple threads to shared resources
Problem of preventing a bad interleaving of operations from different threads

Distributed Computing
Multiple computers communicate through network
Each processor has its own private memory (distributed memory)
Executing sub-tasks on different machines and then merging the results

Cluster Computing
A set of loosely connected computers that work together so that in many respects they can be viewed as single system
Good price / performance, memory not shared

Grid Computing
Federation of computer resources from multiple locations to reach a common goal
Grids tend to be more loosely coupled, heterogeneous, and geographically dispersed

Moore's Law
Doubling of the number of transistors on integrated circuits roughly every two year

Computer Hardware Trend
No more hidden parallelism to be found (ILP)
Transistor number still rising
Clock speed flattening sharply (power consumption, heat generation)
Minimized wire lengths and interconnect latencies

Symmetric MultiProcessor System
Two or more identical processors are connected to a single shared memory
Most common multiprocessor systems today use an SMP architecture

GPGPU
General Purpose Graphical Processing Unit

Overhead of Parallelism
Cost of starting a thread or process
Cost of communicating shared data
Cost of synchronizing
Extra (redundant) computation

Load Imbalance
Some processors are idle due to insufficient parallelism / unequal size tasks

Single/Multiple Instruction Single/Multiple Data

Decomposition – Assignment – Orchestration – Mapping

Decomposition — coverage
Break up computation into tasks to be divided among processes
Identify concurrency and decide level at which to exploit it
Domain – data / Functional – computation, problem

Assignment — granularity
Assign tasks to threads (static/dynamic)
Balance workload, reduce communication and management cost
Together with decomposition, also called partitioning

Orchestration — locality
Structuring (reduce) communication and synchronization
Organizing data structures in memory (reserve locality) and scheduling tasks temporally

Mapping — locality
Mapping threads to execution units (cores) (OS job)
Place related threads on the same processor
Maximize locality, data sharing, minimize costs of comm/sync

Coverage: Amdahl's Law
Potential program speedup is defined by the fraction of code that can be parallelized

Scalability
Capability of a system to increase total throughput under an increased load when resources (typically hardware) are added

Granularity
Qualitative measure of the ratio of computation to communication
Extent to which a system is broken down into small parts

Coarse grained
Relatively large amounts of computational work done between communication events
Consists of fewer, larger components than fine-grained systems
Regards large subcomponents
High computation to communication ratio
More opportunity for performance increase
Advantageous (overhead = comm+sync)

Fine grained
Relatively small amounts of computational work done between communication events
Regards smaller subcomponents of which the larger ones are composed
Low computation to communication ratio
Less opportunity for performance increase
High communication overhead

Load Balancing
Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time
Minimization of task idle time

Static Load Balancing
Programmer make decision and assign fixed amount of work to each core
Low run time overhead
Homogeneous multicores

Dynamic Load Balancing
When one core finishes its allocated work, it takes work from a work queue or core with the heaviest workload
Adapt partitioning at run time to balance load
High runtime overhead
Ideal for codes where work is uneven, unpredictable, heterogeneous multicore

Communication
With message passing, programmer has to understand the computation and orchestrate the communication accordingly
Point to Point / Broadcast + Reduce / All to All / Scatter + Gather
Inter-task communication virtually aways implies overhead
Require some type of synchronization between tasks -> waiting
Latency vs Bandwidth / Synchronous vs Asynchronous / Blocking vs Non-blocking

Message Passing Library
Not a language or compiler specification, specific implementation or product

Synchronization
Coordination of simultaneous events (threads / processes) in order to obtain correct runtime order and avoid unexpected condition

Barrier
Any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier

Lock / Semaphore
First task acquires the lock – safely access the protected data/code
Other tasks attempt to acquire lock but must wait until the task that owns the lock releases it

Non-Uniform Access
Physically partitioned but accessible by all
Same address space
Placement of data affects performance
Cache-Coherent NUMA

Cache Coherence
Uniformity of shared resource data that ends up stored in multiple local caches
When processor modifies shared variable in local cache, different processors may have different value -> need to take actions to ensure visibility or cache coherence

Shared Memory Architecture
Global address space provides user-friendly programming perspective to memory
Fast and uniform data sharing between tasks (due to proximity of memory to CPUs)
Lack of scalability between memory and CPUS
Programmer responsibility for synchronization
Difficult and expensive to design/produce

Distributed Memory Architecture
Scalable (processors, memory)
Cost effective
Programmer responsibility of data communication
No global memory access
Non-uniform memory access time

Thread
Single sequential flow of conrol within a program
Program counter + Register set + Stack
Share - Code section + Data section + OS resources (file)

yield()
release the right of cpu, allows the scheduler to select another runnable thread
join()
one thread can wait for another thread to end
wait()
current thread blocked, placed into wait set, lock object released
notify()
one thread removed from wait set, retain lock for object, resumed from waiting status

Condition Variable
Synchronization primitives that enable threads to wait until a particular condition occurs
Enable threads to atomically release a lock and enter the sleeping state
Without condition variables, programmer need to have threads continually polling
Always used in conjunction with mutex lock

Producer-Consumer Problem
Producer thread do some work and enqueue result objects
Consumer thread dequeue objects and de next stage
Must synchronize access to queue

Mutual Exclusion
Prevent more than one thread from accessing critical section at given time