

Parallel Computing

Basics

Parallel computing

- is a form of computation in which many calculations are carried out simultaneously operating on the principle that
 - large problems can often be divided into smaller ones,
 - which are then solved concurrently ("in parallel").
- There are several different forms of parallel computing:
 - bit-level
 - instruction level
 - data
 - task parallelism.
- Parallelism has been employed for many years, mainly in high-performance computing
- interest in it has grown lately due to the physical constraints preventing frequency scaling.

Versions

- As power consumption (and therefore heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.
- Parallel computers can be roughly classified according to the level at which the hardware supports parallelism,
 - with **multi-core** and **multi-processor** computers having multiple processing elements within a single machine,
 - while **clusters**, **MPPs**, and **grids** use multiple computers to work on the same task.
- Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

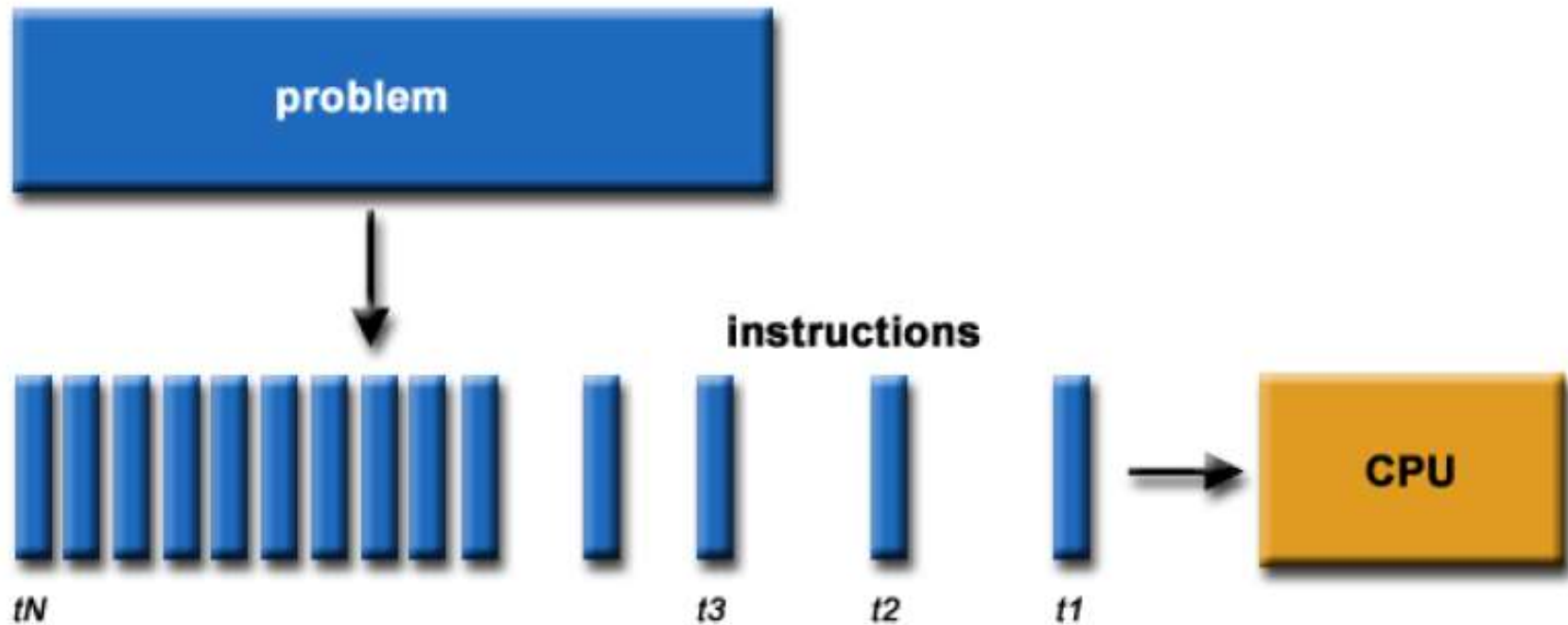
Parallel vs Serial

- **Parallel computer programs** are more difficult to write than **sequential ones**,
 - concurrency introduces several new classes of potential software bugs
 - of which **race conditions** are the most common.
- **Communication** and **synchronization** between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.
- The maximum possible speed-up of a single program as a result of parallelization is known as **Amdahl's law**.

Serial Computation

- **Traditionally**, computer software has been written for **serial computation**.
- To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions.
 - These instructions are executed on a central processing unit on one computer.
 - Only **one instruction may execute at a time**—after that instruction is finished, the next is executed.

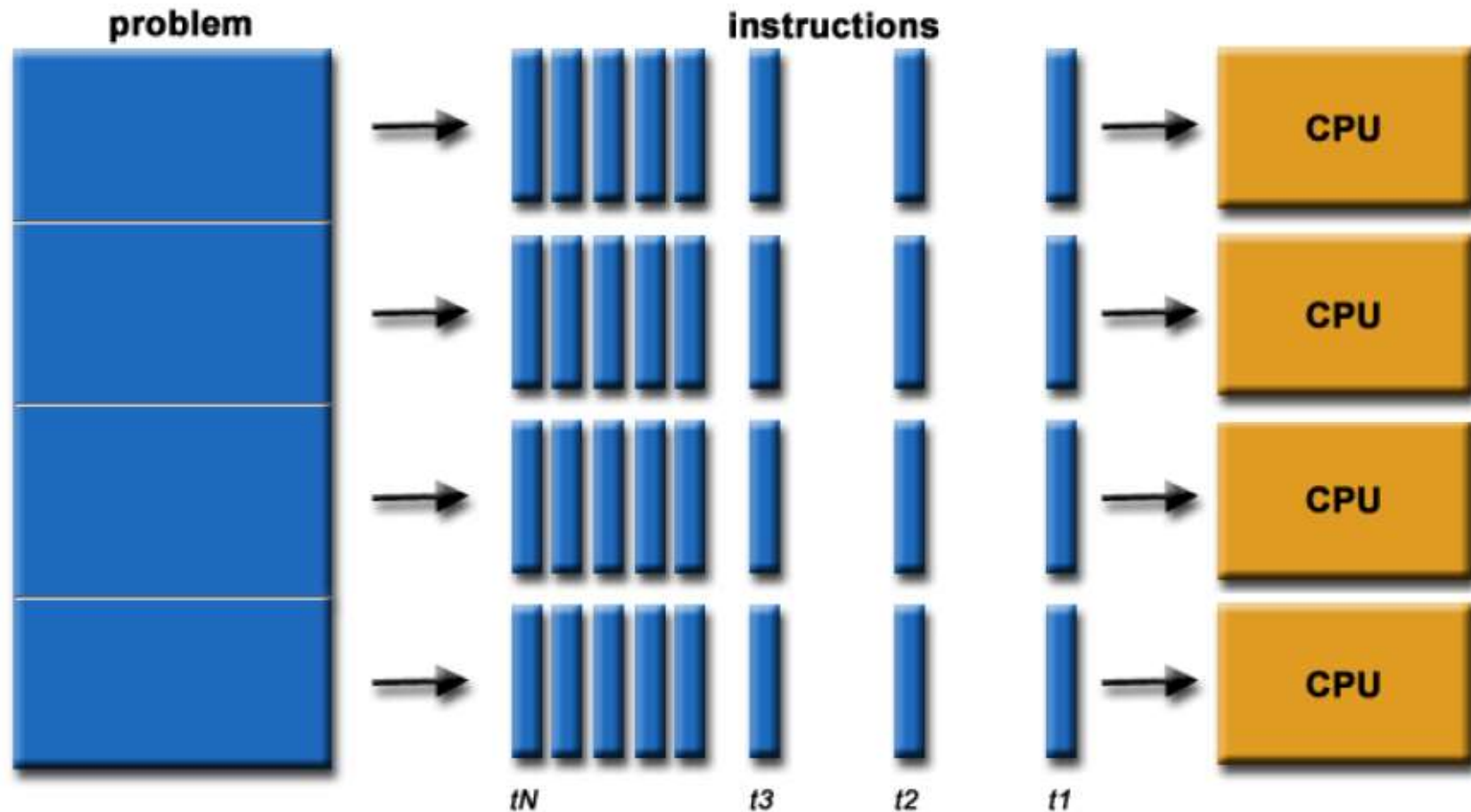
Serial Computation



Parallel Computing

- Parallel computing uses multiple processing elements simultaneously to solve a problem.
- This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others.
- The processing elements can be diverse and include resources such as
 - a single computer with multiple processors,
 - several networked computers,
 - specialized hardware
 - any combination of the above.

Parallel Computing



Why use parallel computing?

- Save time and/or money
- Enables the solution of larger problems
- Provide concurrency
- Use of non-local resources

Frequency

- Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004.
- The **runtime of a program** is equal to the number of instructions multiplied by the average time per instruction.
- Maintaining everything else constant, **increasing the clock frequency** decreases the average time it takes to execute an instruction.
- **An increase in frequency thus decreases runtime** for all compute-bound programs.

Power

$$P = fC_pV^2$$

- Increases in frequency increase the amount of power used in a processor.
 - Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.
- **Moore's Law** is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months.
 - Despite power consumption issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

Amdahl's law and Gustafson's law

- Optimally, the speed-up from parallelization would be linear—
 - doubling the number of processing elements should halve the runtime,
 - and doubling it a second time should again halve the runtime.
- However, very few parallel algorithms achieve optimal speed-up.
 - Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

Amdahl's Law

- The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that
 - a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization.
- A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts.

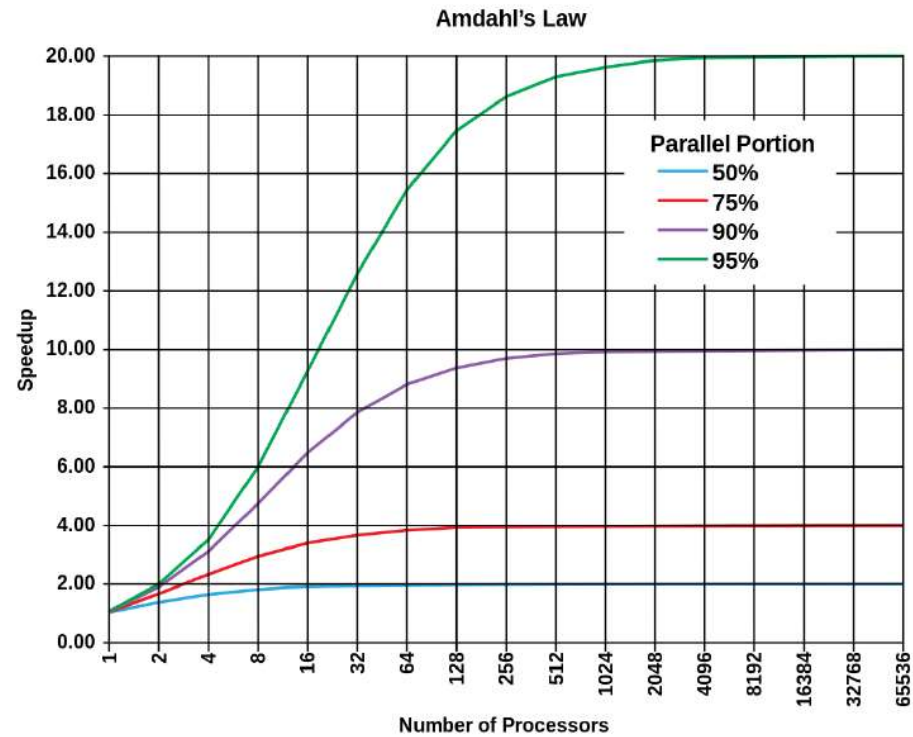
Amdahl's Law

- If α is the fraction of running time a program spends on non-parallelizable parts, and P the number of processors, then the maximum speed-up with parallelization of the program is:

$$MaxSpeedUp = \lim_{p \rightarrow \infty} \frac{1}{\frac{1 - \alpha}{P} + \alpha} = \frac{1}{\alpha}$$

- If the sequential portion of a program accounts for 10% of the runtime ($\alpha = 10\%$), we can get no more than a 10x speed-up, regardless of how many processors are added.
- This puts an upper limit on the usefulness of adding more parallel execution units.

Amdahl's Law



- Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with processors is

$$S(P) = P - \alpha(P - 1) = \alpha + P(1 - \alpha)$$


Example

- Assume that a task has two independent parts, A and B.
 - B takes roughly 25% of the time of the whole

Two independent parts **A** **B**

Original process 

Make **B** 5x faster 

Make **A** 2x faster 

Example

- With effort, a programmer may be able to make this part five times faster,
 - but this only reduces the time for the whole computation by a little.
- In contrast, one may need to perform less work to make part A twice as fast.
- This will make the computation much faster than by optimizing part B,
 - even though B got a greater speed-up (5× versus 2×).

Assumptions

- Both Amdahl's law and Gustafson's law assume that the running time of the sequential portion of the program is independent of P (the number of processors).
- Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also *independent of the number of processors*,
- whereas Gustafson's law assumes that the total amount of work to be done in parallel *varies linearly with the number of processors*.

Parallel Computing

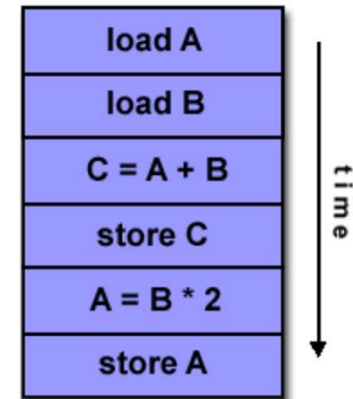
Part 2

Flynn's Taxonomy

S I S D Single Instruction, Single Data	S I M D Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	M I M D Multiple Instruction, Multiple Data

SISD - Single Instruction, Single Data

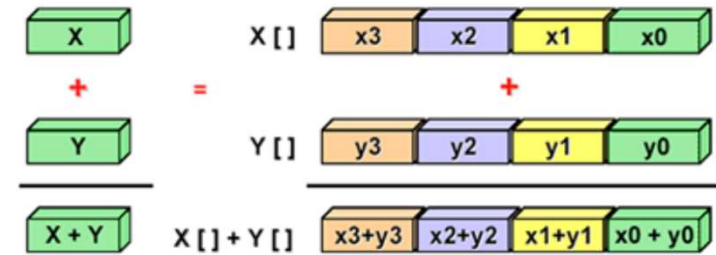
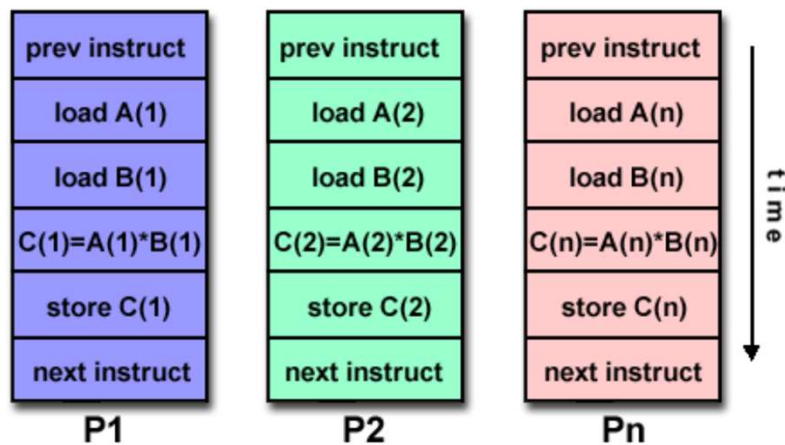
- Serial computer
- Deterministic execution
- Examples:
 - older generation main frames,
 - work stations,
 - PCs



SIMD - Single Instruction, Multiple Data

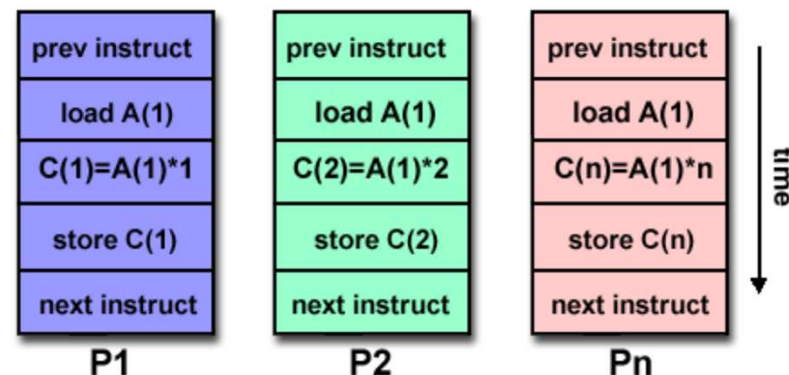
- A type of parallel computer
- All processing units execute the same instruction at any given clock cycle
- Each processing unit can operate on a different data element
- Two varieties:
 - Processor Arrays and
 - Vector Pipelines
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

SIMD - Single Instruction, Multiple Data



MISD - Multiple Instruction, Single Data

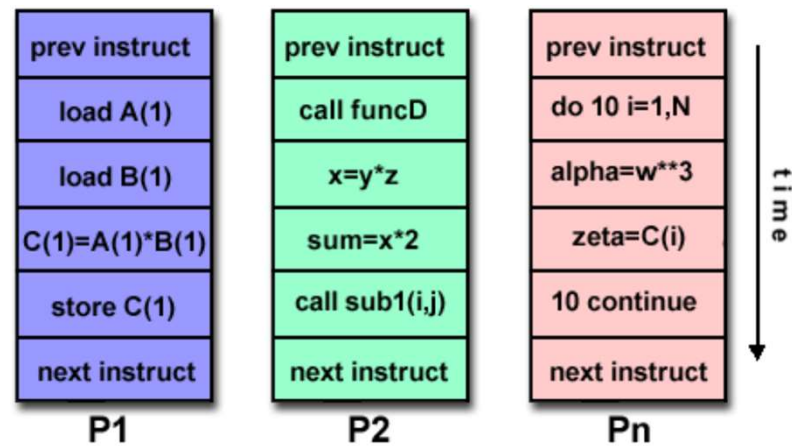
- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples : Carnegie-Mellon C.mmp computer (1971).



MIMD - Multiple Instruction, Multiple Data

- Currently, most common type of parallel computer
 - Every processor may be executing a different instruction stream
 - Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples:
 - most current supercomputers,
 - networked parallel computer clusters and "grids",
 - multi-processor SMP computers,
 - multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components

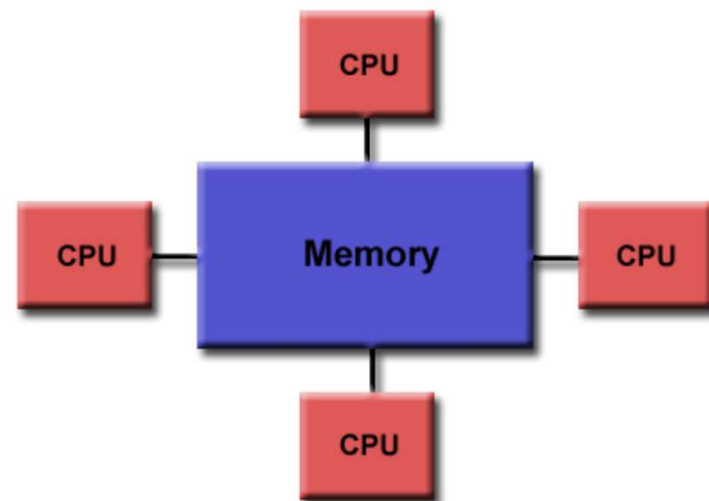
MIMD - Multiple Instruction, Multiple Data



Parallel Computer Architectures

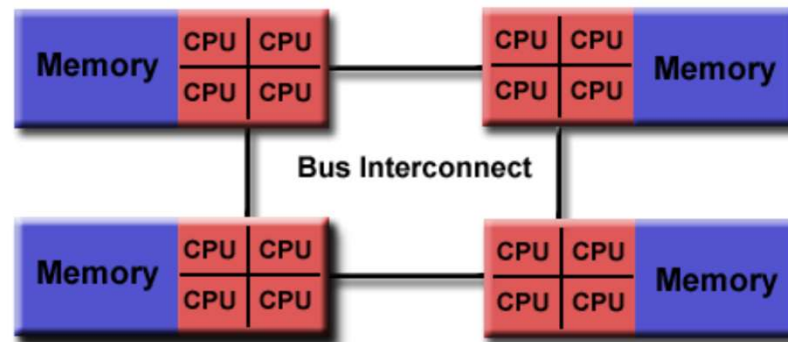
Shared memory / UMA

- Shared memory:
 - all processors can access the same memory
- Uniform memory access (UMA):
 - identical processors-equal access and access times to memory



Laboratory Non-uniform memory access (NUMA)

- all processors have equal access to all memories
- Memory access across link is slower

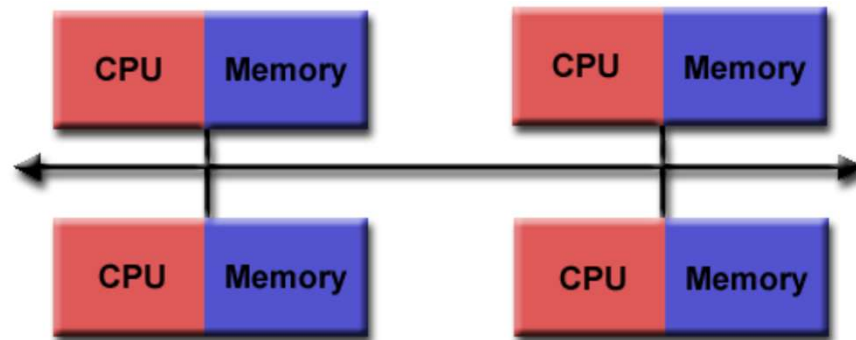


Laboratory Non-uniform memory access (NUMA)

- Advantages:
 - user-friendly programming perspective to memory
 - fast and uniform data sharing due to the proximity of memory to CPUs
- Disadvantages:
 - lack of scalability between memory and CPUs.
 - Programmer responsible to ensure "correct" access of global memory
 - Expense

Distributed memory

- Distributed memory systems require a communication network to connect inter-processor memory.

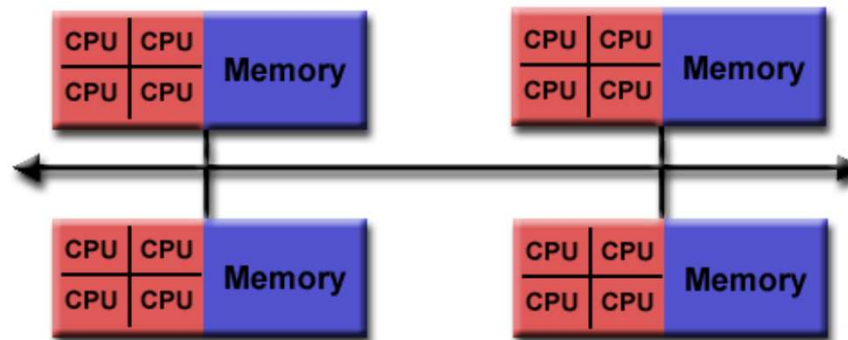


Distributed memory

- Advantages:
 - Memory is scalable with number of processors.
 - No memory interference or overhead for trying to keep cache coherency.
 - Cost effective
- Disadvantages:
 - programmer responsible for data communication between processors.
 - difficult to map existing data structures to this memory organization.

Hybrid Distributed-Shared Memory

- Generally used for the currently largest and fastest computers
- Has a mixture of previously mentioned advantages and disadvantages



Parallel programming models

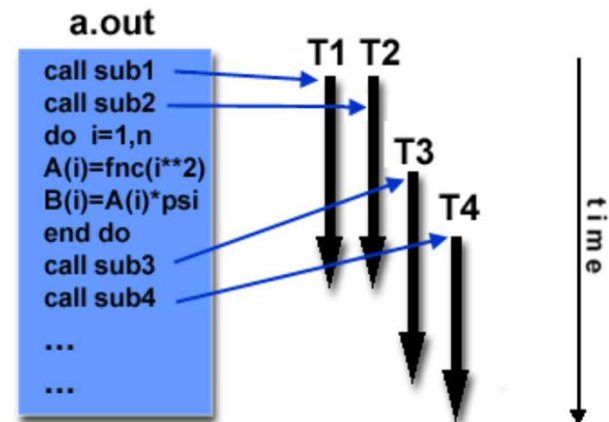
- Shared memory
 - Threads
 - Message Passing
 - Data Parallel
 - Hybrid
-
- All of these can be implemented on any architecture.

Shared memory

- tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- Advantage:
 - no need to explicitly communicate of data between tasks -> simplified programming
- Disadvantages:
 - Need to take care when managing memory, avoid synchronization conflicts
 - Harder to control data locality

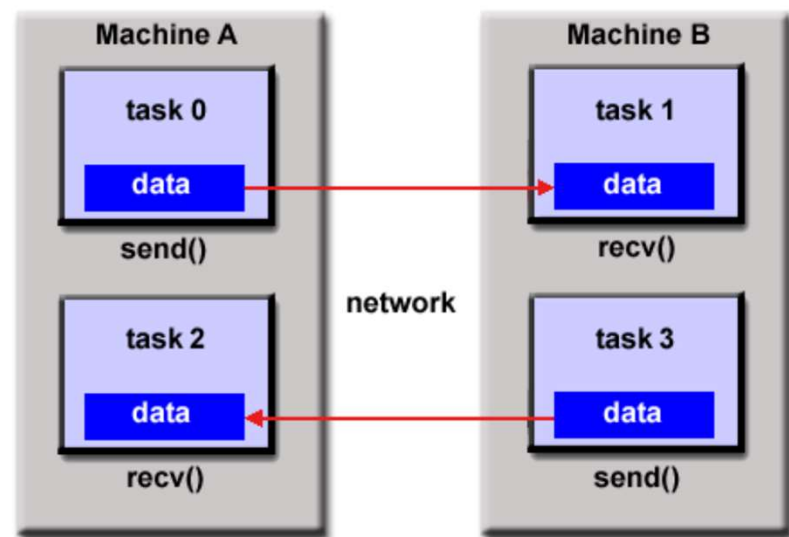
Threads

- A thread can be considered as a subroutine in the main program
- Threads communicate with each other through the global memory
- commonly associated with shared memory architectures and operating systems



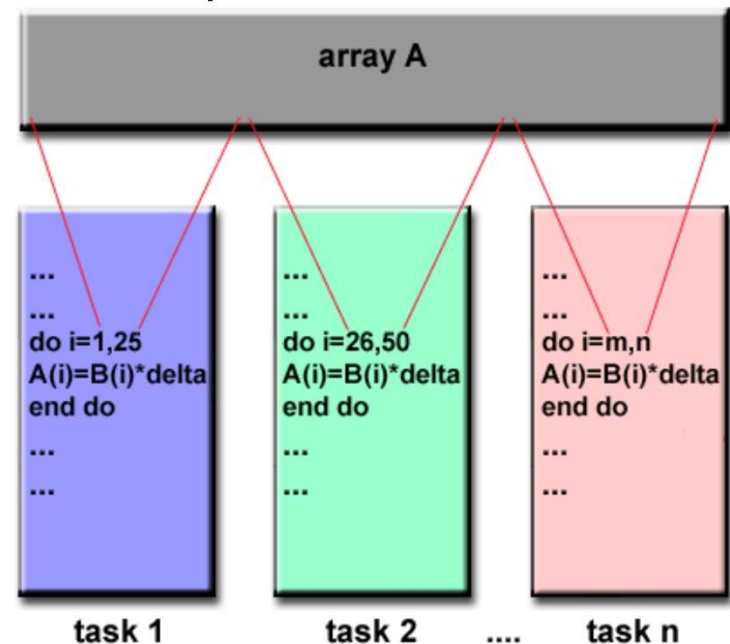
Message Passing

- A set of tasks that use their own local memory during computation.
- Data exchange through sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
- MPI (released in 1994)
- MPI-2 (released in 1996)



Data Parallel

- The data parallel model demonstrates the following characteristics:
 - Most of the parallel work performs operations on a data set, organized into a common structure, such as an array
 - A set of tasks works collectively on the same data structure, with each task working on a different partition
 - Tasks perform the same operation on their partition



Data Parallel

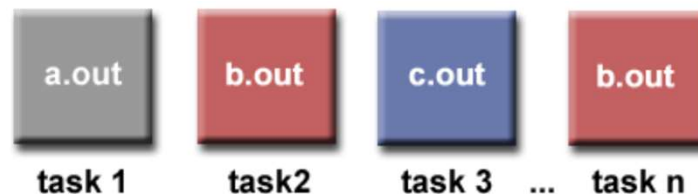
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Other Models

- Hybrid
 - combines various models, e.g. MPI/OpenMP
- Single Program Multiple Data (SPMD)
 - A single program is executed by all tasks simultaneously

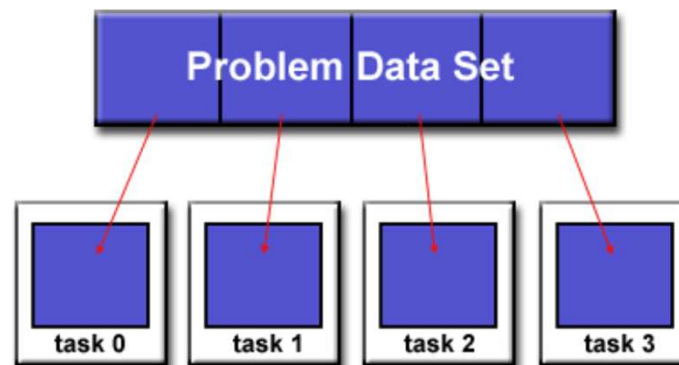


- Multiple Program Multiple Data (MPMD)
 - An MPMD application has multiple executables.
 - Each task can execute the same or different program as other tasks.



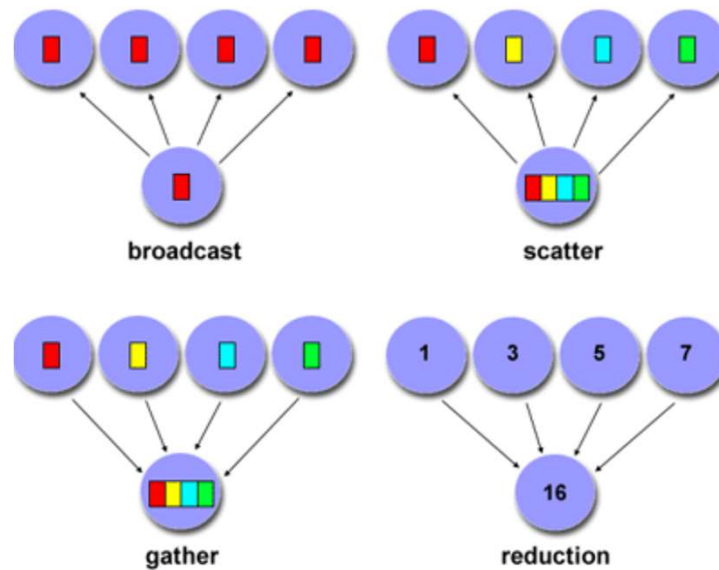
Designing Parallel Programs

- Examine problem:-
 - Can the problem be parallelized?
 - Are there data dependencies?
 - where is most of the work done?
 - identify bottlenecks (e.g. I/O)
- Partitioning
 - How should the data be decomposed?



Communications

- Types of communication:
 - point-to-point
 - collective



Synchronization types

- Barrier
 - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
 - When the last task reaches the barrier, all tasks are synchronized.
- Lock / semaphore
 - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
 - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
 - Can be blocking or non-blocking