MICROPROCESSOR AND COMPUTER ARCHITECTURE

UNIT-5

advances in architecture

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

ADVANCED ARCHITECTURE

- M1 — system on chip, ARM (RISC) architecture, levels of cache
- Cache levels exist even for SOC
- 8 cores, 8 GPUs on chip

- Multicore, multiprocessing, parallel computing

High Performance Computing

- Efficient algorithms on computers
- Capable of highest performance
- To solve most demanding problems

1) Higher Speed
2) Higher Throughput
3) High Computational Power

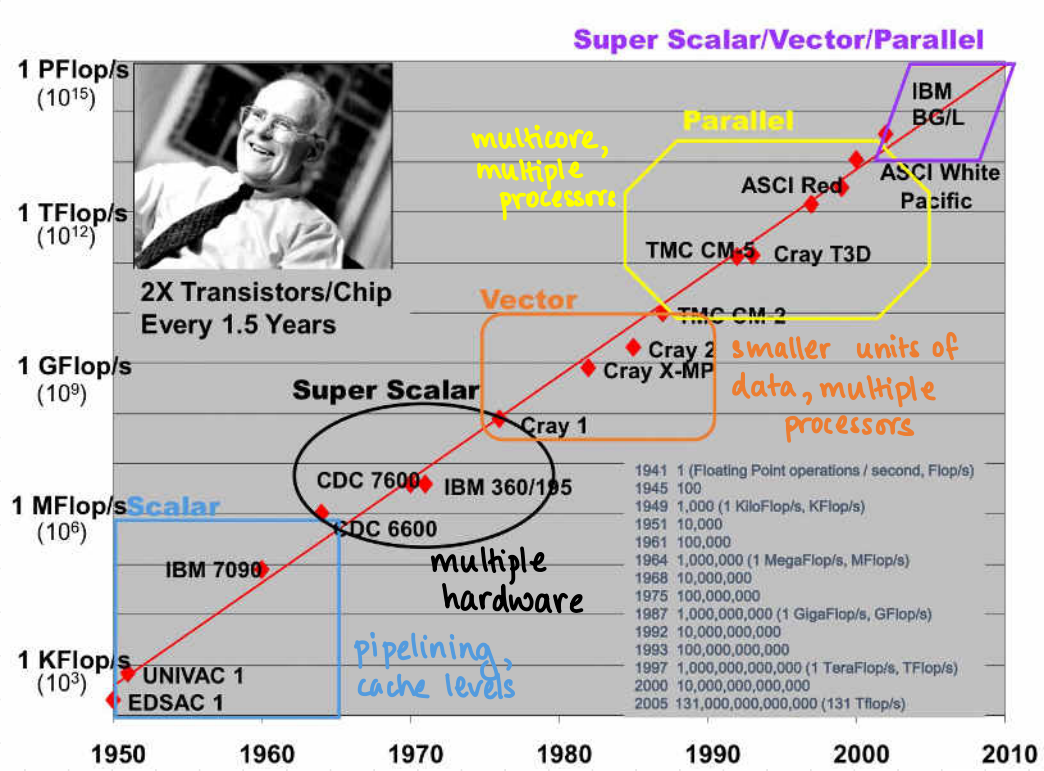
Quantifying Capability to Solve Problem

- Floating Point Operations Per Second

Name	Unit	Value
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}

Timeline of Growth

Blue Gene



Moore's Law

- No. of transistors / chip doubles every 2 years
- Cost of computer halved

EQUATION

$$P_n = P_0 \times 2^n$$

- P_n: processing power in future years
- P_0: processing power in beginning year
- n: number of years to develop a new microprocessor divided by 2 (no. of 2-years)

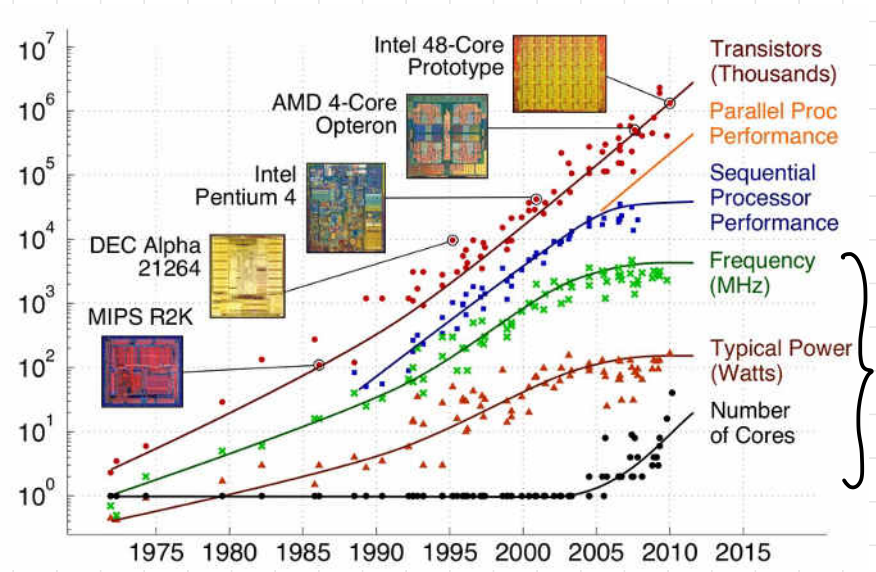
Q: In 1988, the number of transistors in the Intel 386 SX microprocessor was 275,000. What were the transistors counts of the Pentium II Intel microprocessor in 1997?

$$n = \frac{1997-1988}{2} = \frac{9}{2}$$

$$P_0 = 275000$$

$$P_n = 275000 \times 2^{4.5} = 6222539$$
$$= 6.2 \text{ million}$$

Growth & Change in Trend



Denard scaling: as transistors get smaller, power density stays constant

Shift to Parallel Processing

1) Memory Wall Challenge

- Gap between processor and memory performances
- Gap increasing
- Memory latency and bandwidth insufficient; acts as bottleneck
- Processors stall

2) Power Wall Challenge

- Power delivery and dissipation
- Difficulty in scaling performance of chips and systems
- Faster computers get very hot

AMDAHL'S LAW*

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{s}}$$

s = speedup factor

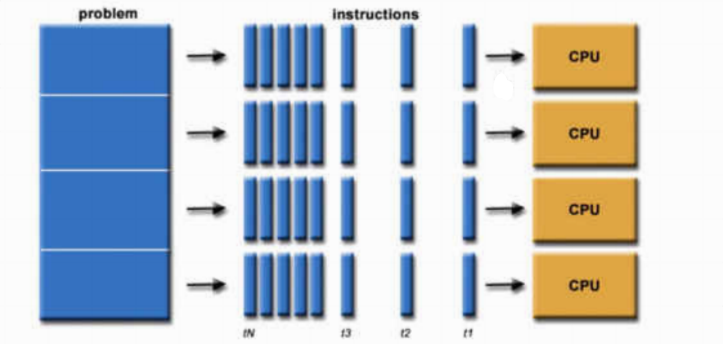
f = fraction of program that can be optimised

$1-f$ = fraction of program that cannot be optimised

* more on page 39

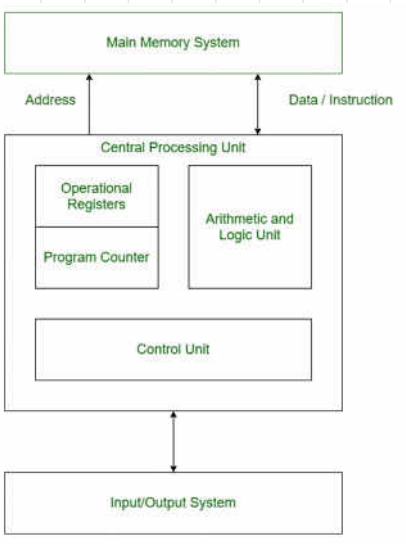
Parallel computing

- Multiple CPUs for single program



von Neumann Architecture

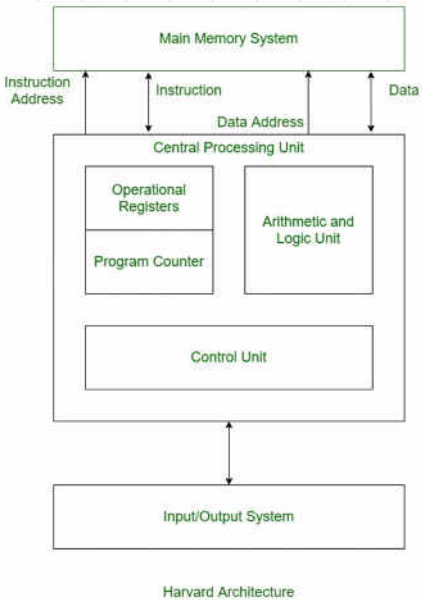
- Sequence of instructions stored in memory
- Executed sequentially
- Stored-program concept : sequence of read-write operations on memory
- Data and instructions both stored in a single memory



Von Neumann Architecture

Harvard Architecture

- Separate storage and buses for instructions and data (modern)
- Can fetch and load/store at the same time (pipelining)



Harvard Architecture

Shift to Parallel Computing

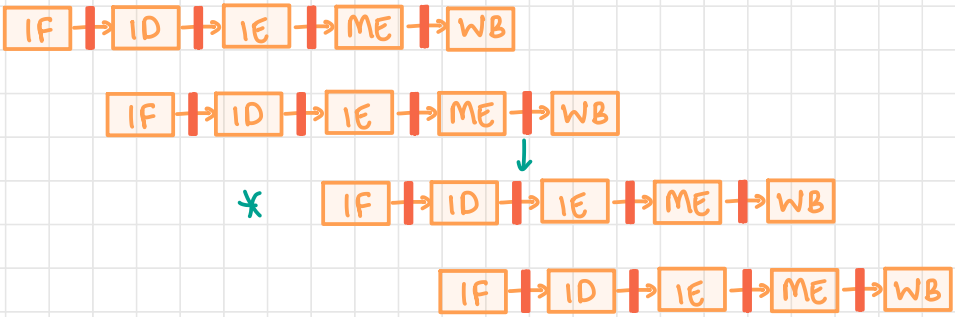
1. Bit-Level Parallelism
   - 8-bit processor to add 16-bit numbers

2. Instruction-Level Parallelism
   - Pipelining (different stages)

3. Loop-Level Parallelism
   - No dependency — split across cores
   - Loop unrolling for dependency

- Eg:
```
for (i=1; i<=1000; ++i) {
    x[i] = x[i] + y[i];
}
```

```
load  x[0]
load  y[0]
add   x[0], y[0]
store x[0]
```
} 1000 times, 1 stall



Loop Unrolling

reschedule

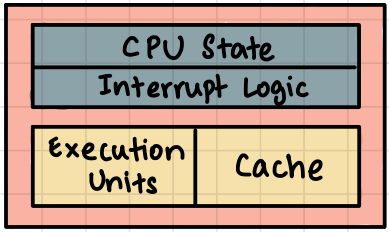
```
load  x[0]
load  y[0]
add   x[0], y[0]
store x[0]
load  x[1]
load  y[1]
add   x[1], y[1]
store x[1]
```

4. Thread-Level Parallelism

- fine-grained thread & coarse-grained thread

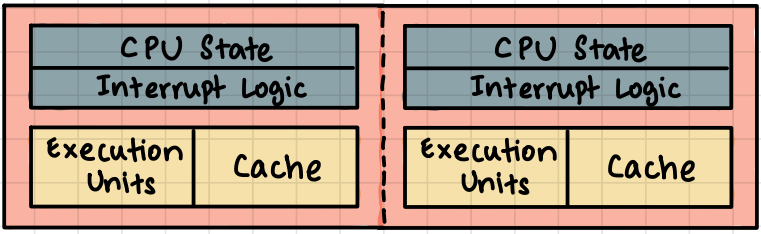
Core

- execution state of program (Reg, PC, stack pointer)
- interrupt logic
- execution units
- cache
- single threaded processor



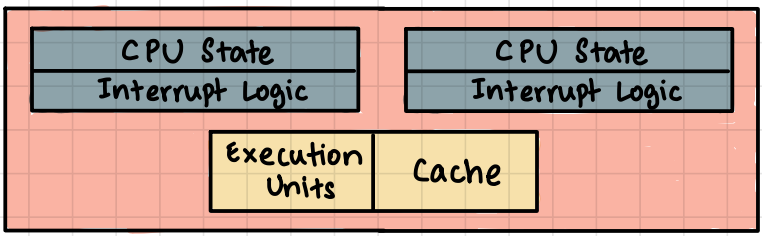
Multicore

- multiple cores on single die / chip
- chip Multiprocessor (CMP)
- thread level & task level parallelism
- each core independently executes a task (or thread)
- cores can share resources
- multithreading and multiprocessing supported



Hyperthreading

- Simultaneous multi threading (SMT)
- thread defined by architecture state (interrupt logic, reg)
- shared execution units and cache
- Intel



- Amdahl's law for HT

$$\text{speedup} = \frac{1}{S + \frac{(1-S)}{0.67n} + H(n)}$$

(a) Fine-Grained Thread

- different parts of program run parallely
- programmers explicitly specify parts of program to run parallely
- hardware extracts parallelism and dynamically schedules
- compiler dynamically schedules

(b) Coarse-Grained Threads

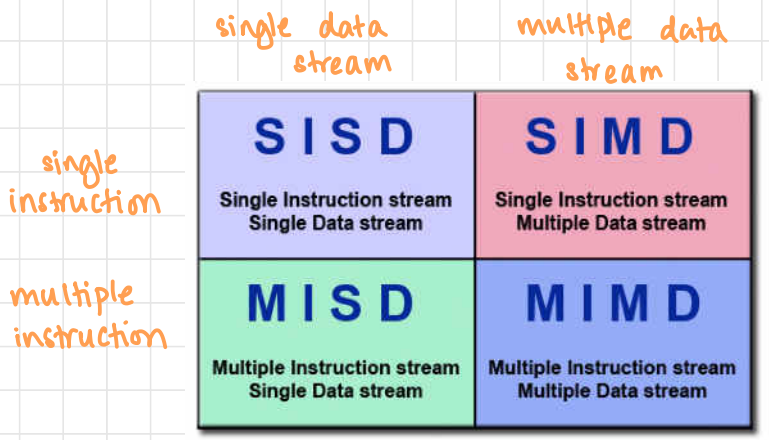
- OS responsible for scheduling tasks on different cores

5. Task-Level Parallelism

- OS or programmer
- processes, tasks, jobs

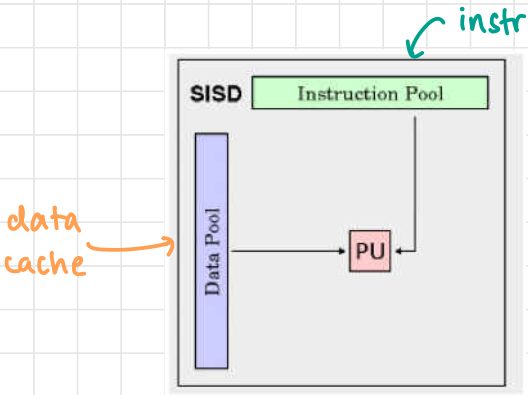
CLASSIFICATION of PARALLEL COMPUTERS

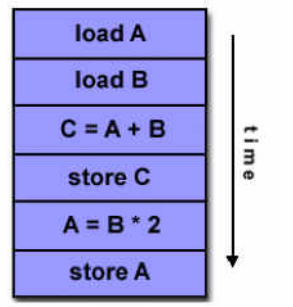
- Flynn's Taxonomy of Computer Architecture
- Two independent dimensions: instructions and data



(1) SISD : Single Instruction, Single Data

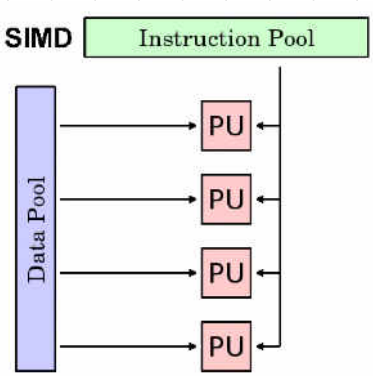
- Single instruction: only one instruction stream being accessed by CPU during single clock cycle
- Single data: only one data stream being accessed by CPU during single clock cycle
- Deterministic
- Intel Atom Family (Silverthorne, Lincroft, Diamondville, Pineview) — rarely found
- Older; sequential execution



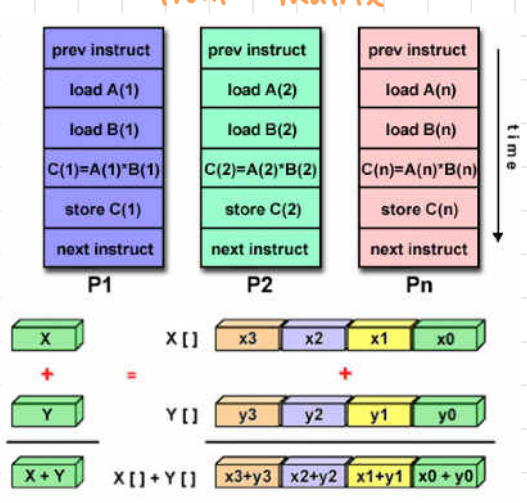


(2) SIMD: Single Instruction, Multiple Data

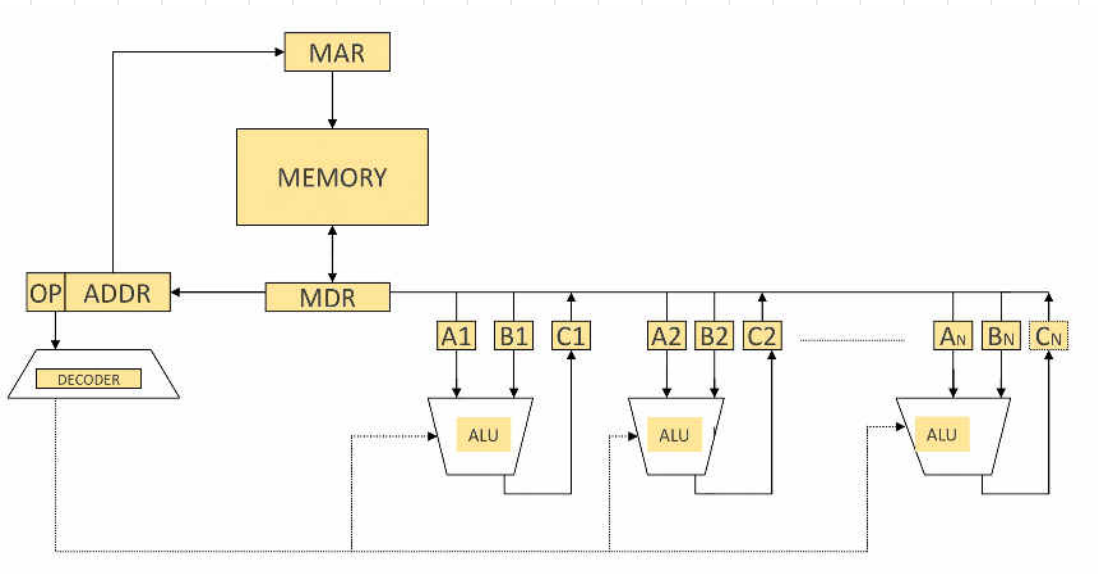
- Specialised problems with high degree of regularity (eg: image processing)
- Two varieties: processor arrays & vector pipelines
- Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2



each core on diff mult from matrix

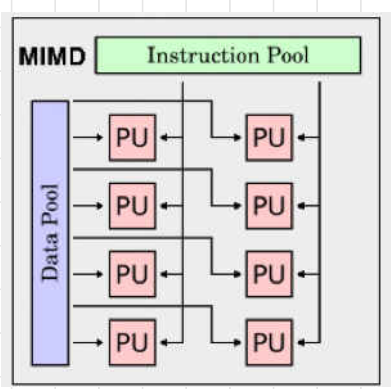


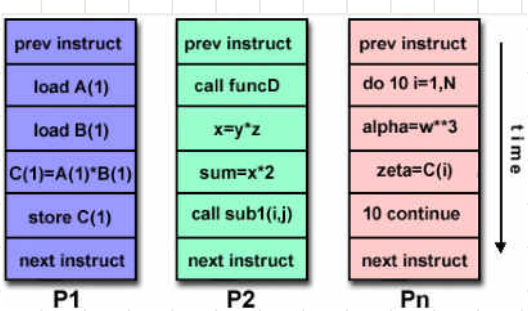
Array Processor



(3) MIMD : Multiple Instruction, Multiple Data

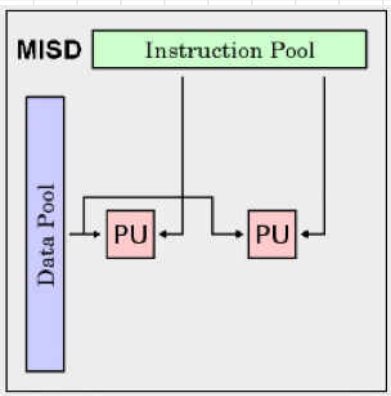
- Most common type of parallel computer
- Synchronous or asynchronous, deterministic or non-deterministic execution

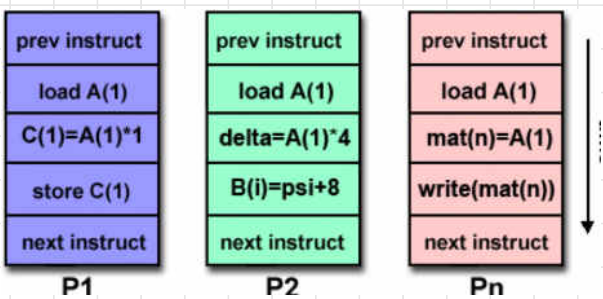




(4) MISD: Multiple Instruction, Single Data

- Few exist in real life (experimental one at CMU); practical purposes — does not exist
- Single data stream fed into multiple processing units
- Systolic arrays





MODERN CLASSIFICATION

- Parallelism can be achieved in two ways

(1) Data Parallelism

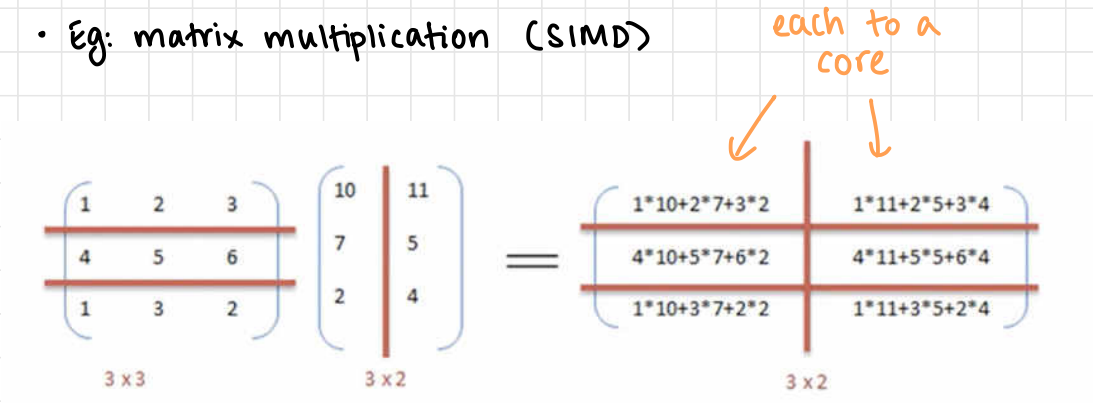
- operating on multiple data in parallel

(2) Function Parallelism

- performing many functions in parallel (control parallelism, task parallelism)

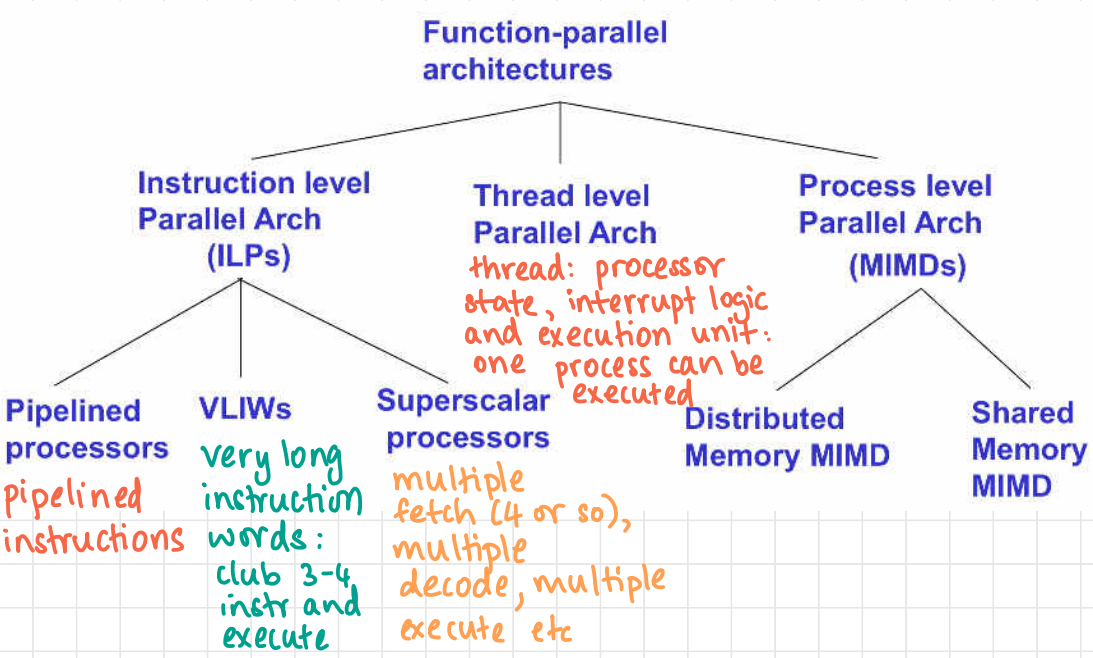
DATA PARALLELISM

- Eg: matrix multiplication (SIMD)



FUNCTIONAL PARALLELISM

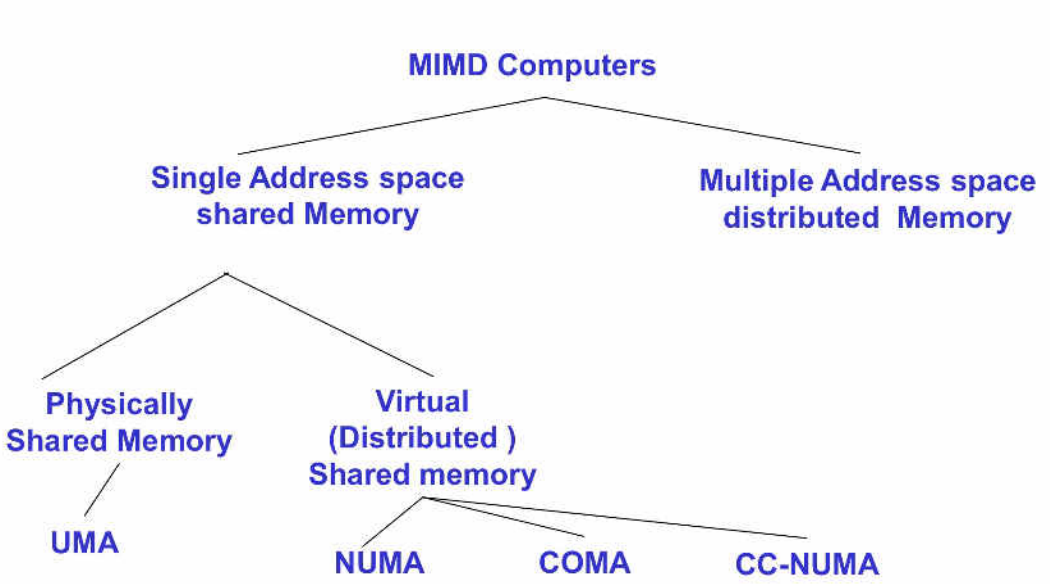
- Functional programming: big task broken into smaller tasks that are independently executed in parallel and later combined to give the final result



UNIX Process

- Consists of address space, large set of process state values, one thread of execution
- Task of kernel: create processes and dispatch them to different CPUs to maximise system utilisation

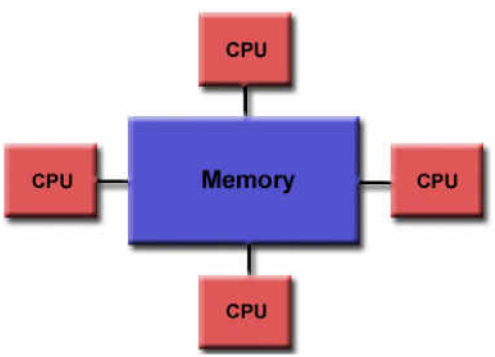
Classification of MMID Computers

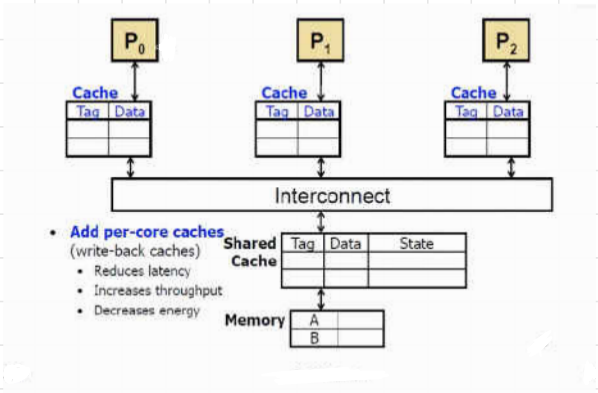


PARALLEL COMPUTER MEMORY ARCHITECTURES

- Shared memory
- Distributed memory
- Hybrid Distributed-Shared Memory
- Note: here memory is cache and not main memory

(a) Shared Memory Architecture

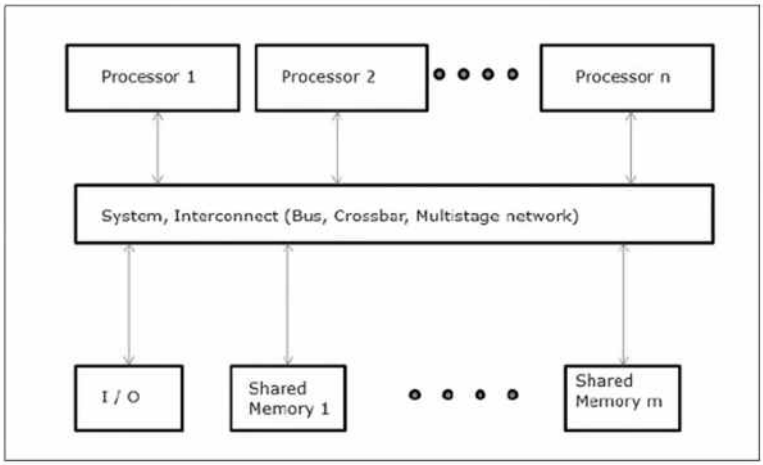




- All CPUs share memory
- Shared memory parallel computers
- All processes can access memory as global address space
- Processors operate independently but share memory resources
- Two classes based on access times: UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access)

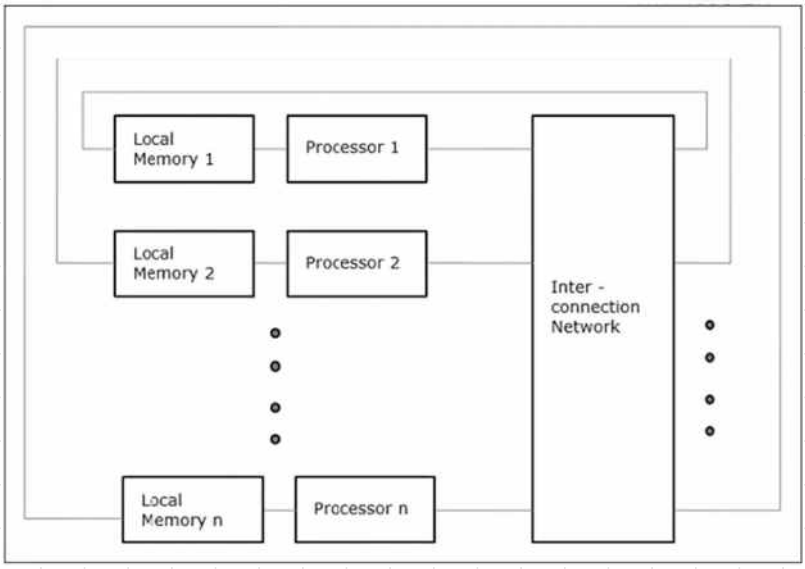
(i) Uniform Memory Access (UMA)

- Today's Symmetric Multiprocessor machines (SMPs)
- same time for all CPUs (processors) to access memory
- Also called CC-UMA (Cache-Coherent UMA)
- If one processor updates location in shared memory (cache), all processors know about update
- Cache coherency accomplished at hardware level



(ii) Non-Uniform Memory Access (NUMA)

- Often: linking 2 or more SMPs
- Memory access time across link is slow
- Cache coherency maintained (CC-NUMA)



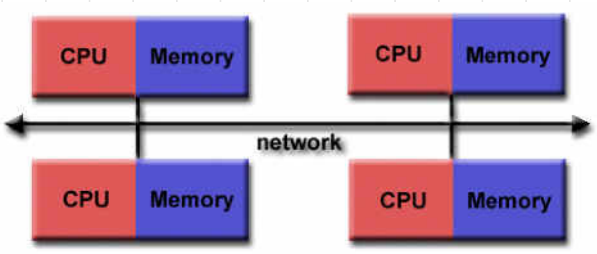
advantages

- Global address space is user friendly for memory
- Fast data sharing between tasks

disadvantages

- Lack of scalability (more CPUs => geometrically more traffic on shared path
- Responsibity on programmer to ensure correct access to global memory
- Expensive to produce for more no. of processors

(b) Distributed Memory Architecture



- Each CPU has its local memory but can access all
- Communication network to connect inter-processor memory
- No global address space; no mapping of memory addresses
- No concept of cache coherency
- Programmer's responsibility to access data from another processor
- Synchronisation between tasks programmer's responsibility (initiate communication, handshake, free network etc.)
- Network fabric varies; can be ethernet

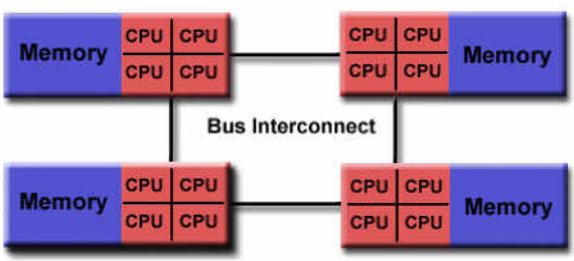
advantages

- Scalable memory with increase in no. of processors
- Rapid access to processor's memory ; no cache coherency overhead
- Cost effective ; can use off-shelf processors and networking

disadvantages

- Programmer responsible for data communication between processors
- NUMA times
- Could be hard to map data structures based on global memory

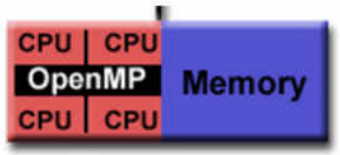
(c) Hybrid Architecture



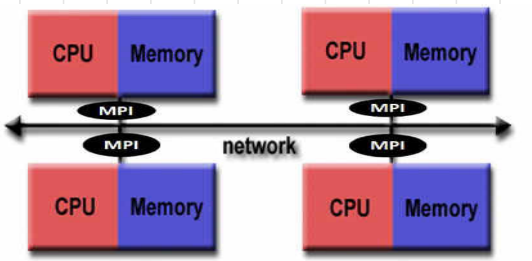
- Shared memory component: cache-coherent SMP machine
- All processors on given SMP access memory as global
- Distributed component is networking of multiple SMPs
- SMPs only know about their memory
- Network communications must transfer data between SMPs

PARALLEL PROGRAMMING LANGUAGES

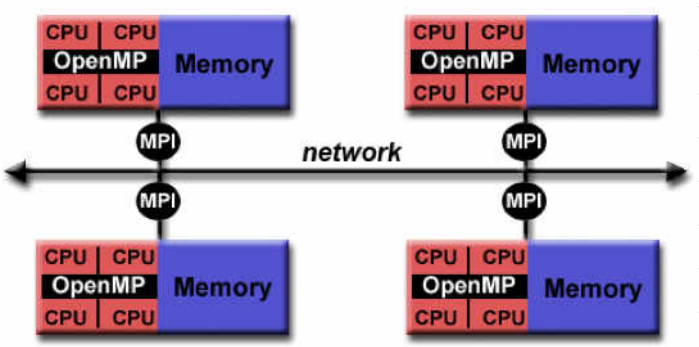
- Shared memory APIs: OpenMP —— C/C++, Fortran, Python



- Distributed memory APIs: MPI (Message Passing Interface) —— C/C++, Fortran, Java, Python, R, Ocaml etc.



- Hybrid memory: combination of OpenMP and MPI



Shared Memory Programming

- OpenMP API in C

```c
#include <omp.h>
#include <stdio.h>

/*
To compile in MacOS: gcc -Xpreprocessor -fopenmp sharedmem.c -lomp -o <output_file>

To compile in Linux: gcc -fopenmp sharedmem.c -o <output_file>
*/

int main(int argc, char** argv) {
    int iam, np;
    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d\n", iam, np);
    }
    return 0;
}
```

Executed in MacOS

```
→  Unit 5 gcc -Xpreprocessor -fopenmp sharedmem.c -lomp -o sharedmem
→  Unit 5 ./sharedmem
Hello from thread 3 out of 8
Hello from thread 0 out of 8
Hello from thread 4 out of 8
Hello from thread 2 out of 8
Hello from thread 6 out of 8
Hello from thread 1 out of 8
Hello from thread 5 out of 8
Hello from thread 7 out of 8
```

Executed in Ubuntu 20

```
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ gcc -fopenmp sharedmem.c -o sharedmem
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ ./sharedmem
Hello from thread 0 out of 2
Hello from thread 1 out of 2
```

Distributed Memory Programming

- MPI API in C

```c
#include <mpi.h>
#include <stdio.h>

/*
Compile on MacOS and Linux: mpicc distmem.c -o <output_file>
Execute on MacOS and Linux: mpirun ./<output_file>
*/

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int numprocs, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello from rank %d out of %d processors\n", rank, numprocs);
    MPI_Finalize();
}
```

Executed on MacOS

```
→  Unit 5 mpicc distmem.c -o distmem
→  Unit 5 mpirun ./distmem
Hello from rank 0 out of 4 processors
Hello from rank 1 out of 4 processors
Hello from rank 2 out of 4 processors
Hello from rank 3 out of 4 processors
```

Executed on Ubuntu

```
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ mpicc distmem.c -o distmem
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ mpirun ./distmem
Hello from rank 0 out of 1 processors
```

Hybrid Programming

- Both OMP and MPI

```c
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

/*
To compile in MacOS: mpicc -Xpreprocessor -fopenmp <filename> -lomp -o <executable>
To compile in Linux: mpicc -fopenmp <filename> -o <executable>

To execute: mpirun <executable>
*/

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int numprocs, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int iam, np;
    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d and rank %d out of %d processor\n", iam, np, rank, numprocs);
    }

    MPI_Finalize();
}
```

Executed in Ubuntu

```
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ mpicc -fopenmp hybrid.c -o hybrid
vibhamasti@ubuntu:~/Personal/CS 4/MPCA$ mpirun ./hybrid
Hello from thread 0 out of 2 and rank 0 out of 1 processor
Hello from thread 1 out of 2 and rank 0 out of 1 processor
```

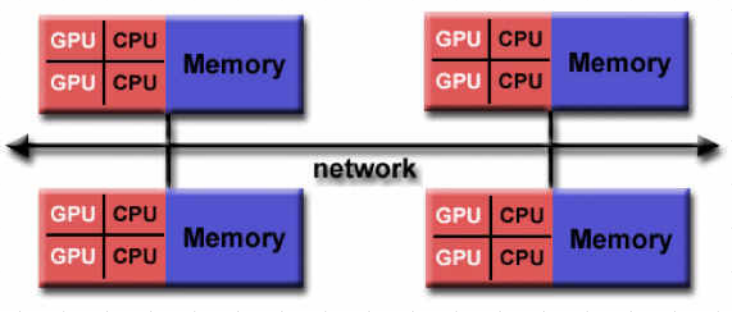
Executed in MacOS

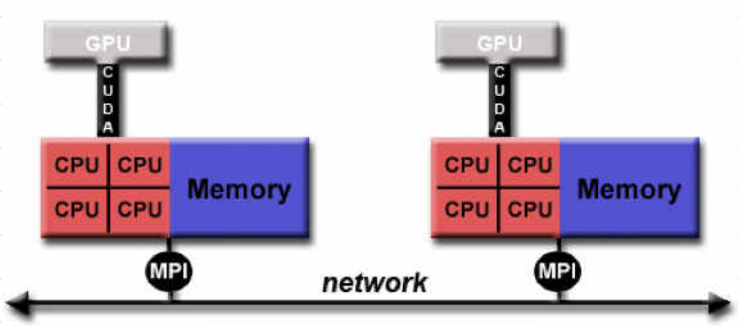
```
➜  Unit 5 mpicc -Xpreprocessor -fopenmp hybrid.c -lomp -o hybrid
➜  Unit 5 mpirun ./hybrid
Hello from thread 0 out of 8 and rank 1 out of 4 processor
Hello from thread 1 out of 8 and rank 1 out of 4 processor
Hello from thread 2 out of 8 and rank 1 out of 4 processor
Hello from thread 4 out of 8 and rank 1 out of 4 processor
```

⋮ (truncated)

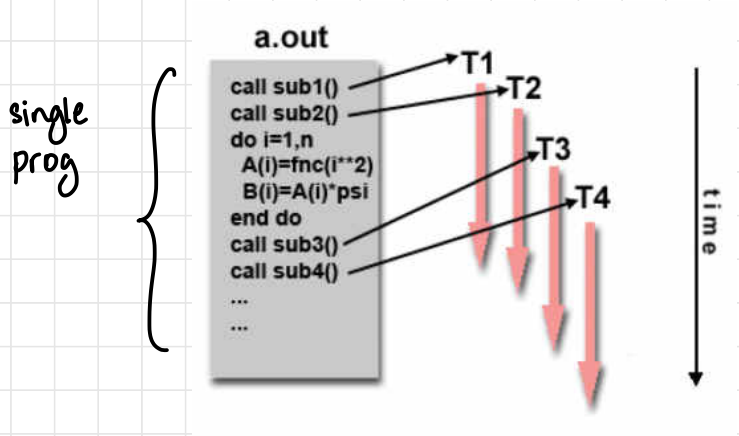
WITH GPU

- Cannot use only OMP & MPI
- CUDA

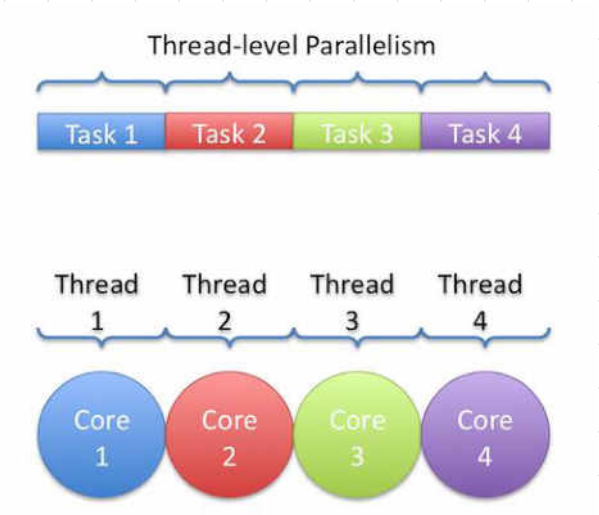




Single Program Multiple Data



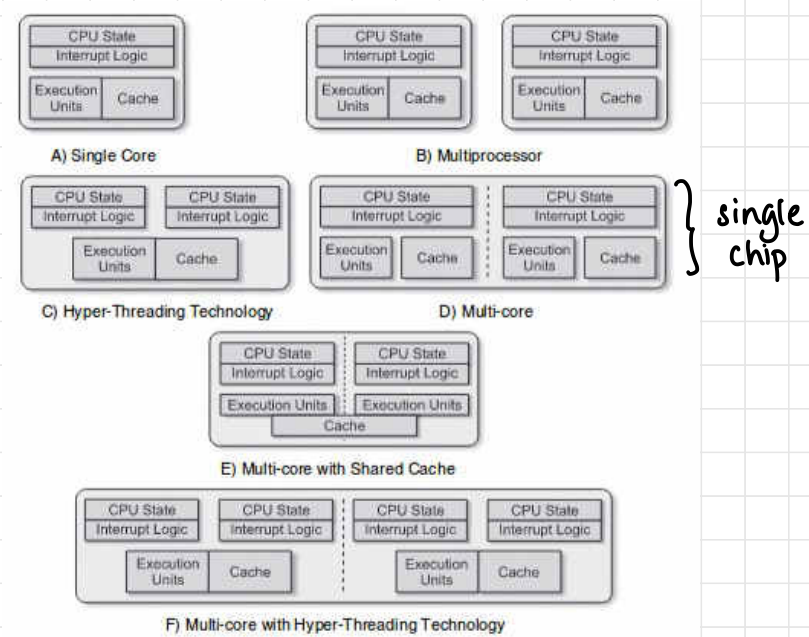
Multiple Program Multiple Data



Architectural Innovations for Improved Performance

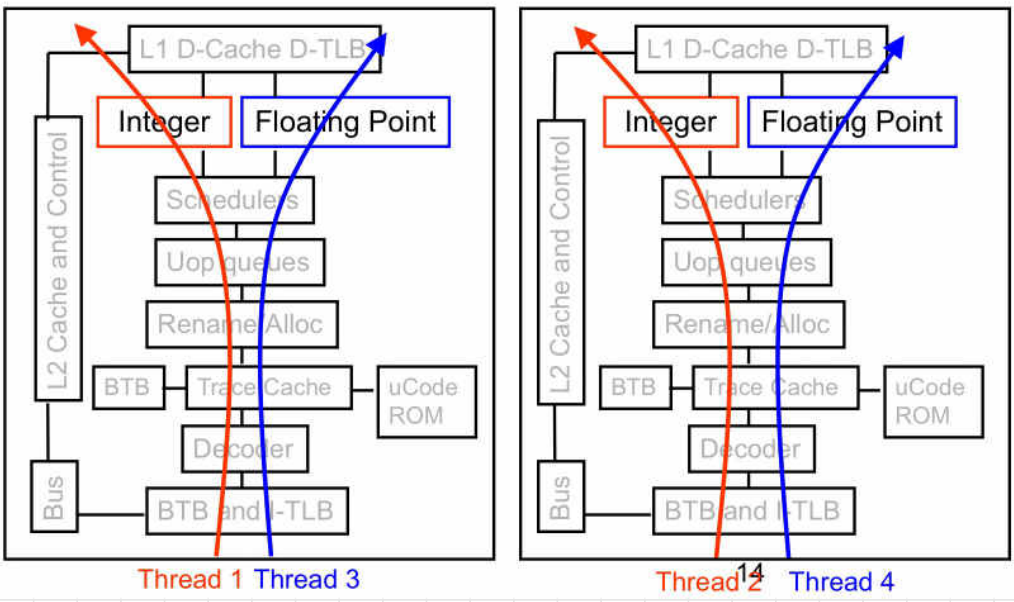
	Architectural method	Improvement factor
Established methods	1. Pipelining (and superpipelining)	3-8
	2. Cache memory, 2-3 levels	2-5
	3. RISC and related ideas	2-3
	4. Multiple instruction issue (superscalar)	2-3
	5. ISA extensions (e.g., for multimedia)	1-3
Newer methods	6. Multithreading (super-, hyper-)	2-5 ?
	7. Speculation and value prediction	2-3 ?
	8. Hardware acceleration	2-10 ?
	9. Vector and array processing	2-10 ?
	10. Parallel/distributed computing	2-1000s ?

Different Types of Multicore Architecture



Thread - Level Parallelism

- SMT Dual-Core

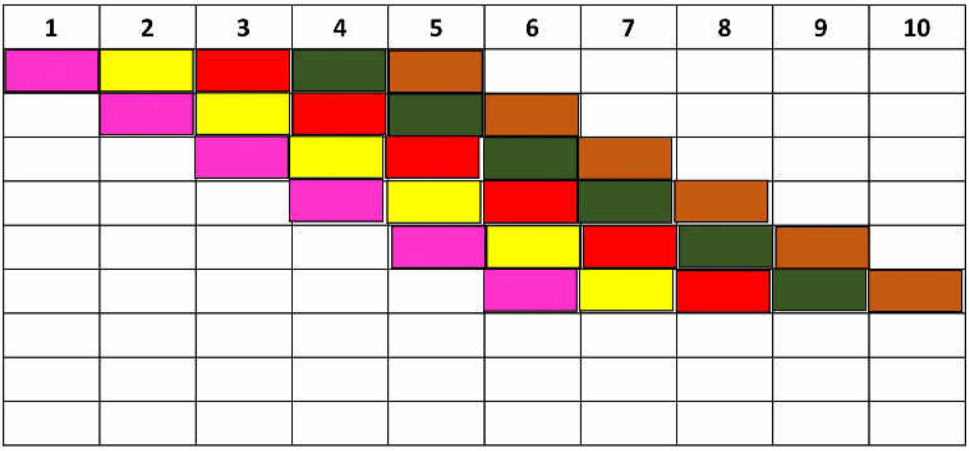


Instruction - Level Parallelism

(i) Scalar

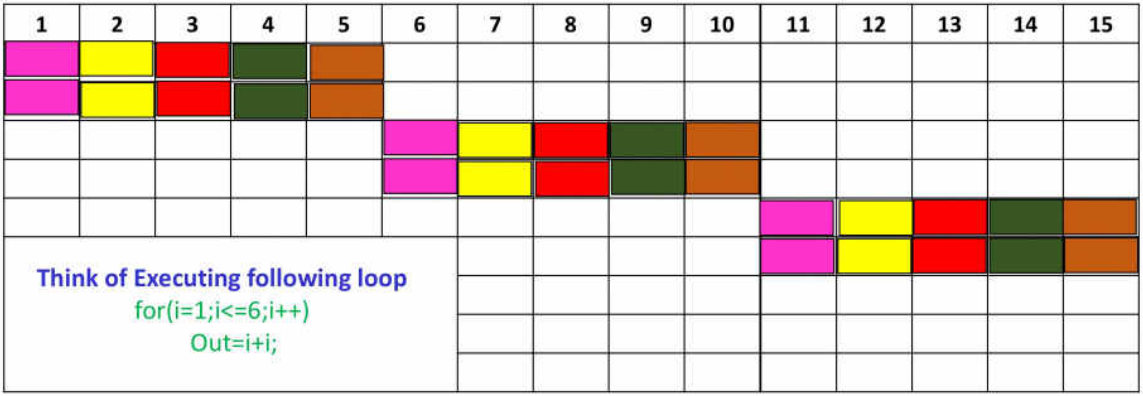
1	2	3	4	5	6	7	8	9	10

(ii) Pipelining



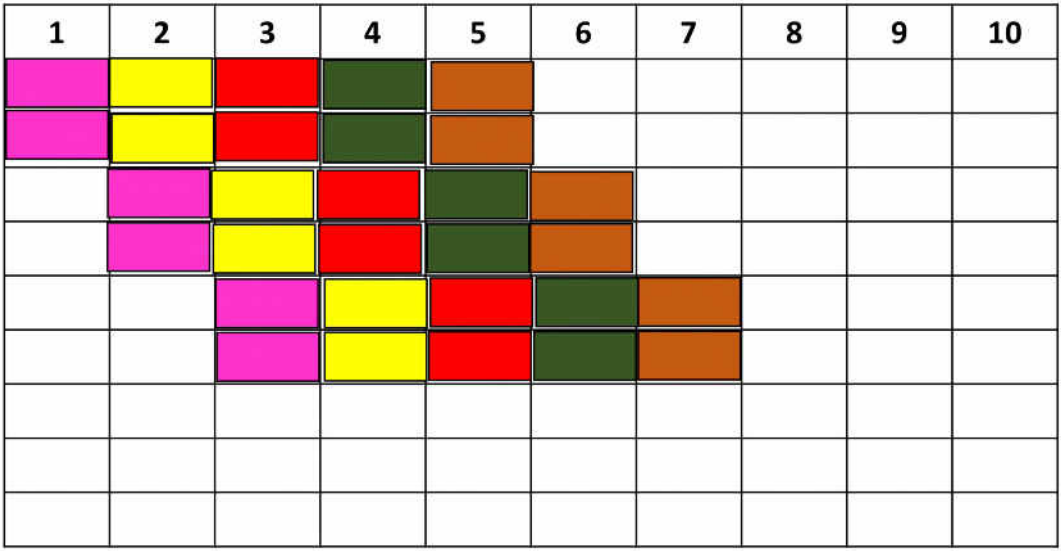
(iii) Super-Scalar

- multiple fetch at once ; more than 1 inst per CC
- parallel execution units

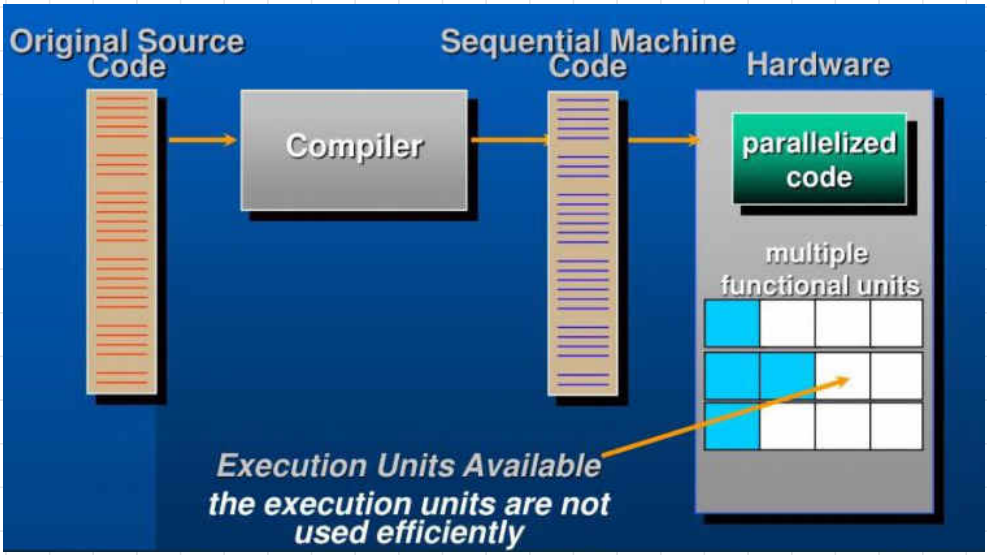


(iv) Super Pipelining

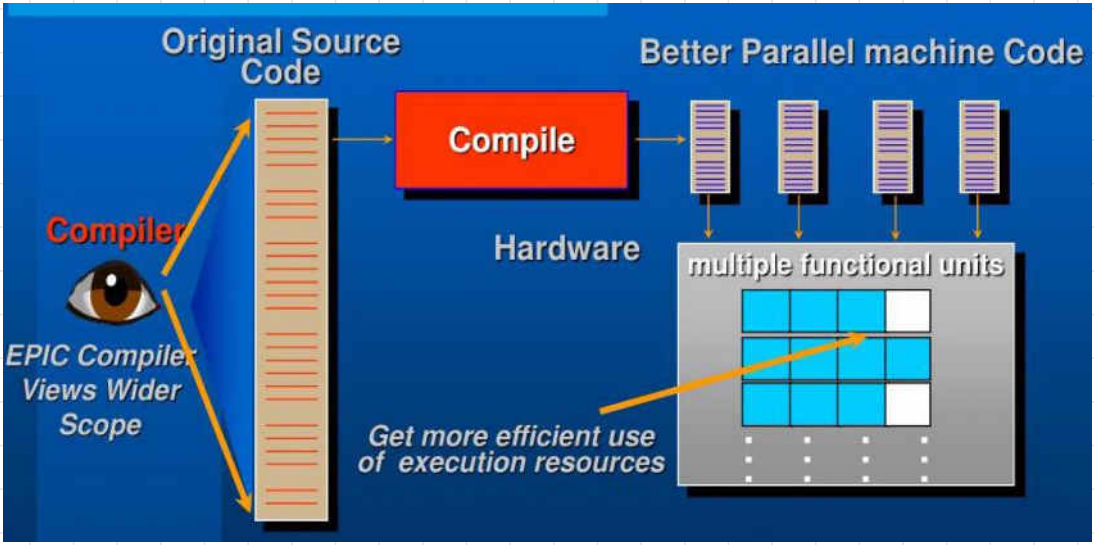
- increase depth (more pipeline stages)
- 0.5 cc for one stage or 2 stages per cc, here



Dynamic Parallelism (Hardware)



Static Parallelism (Compiler)



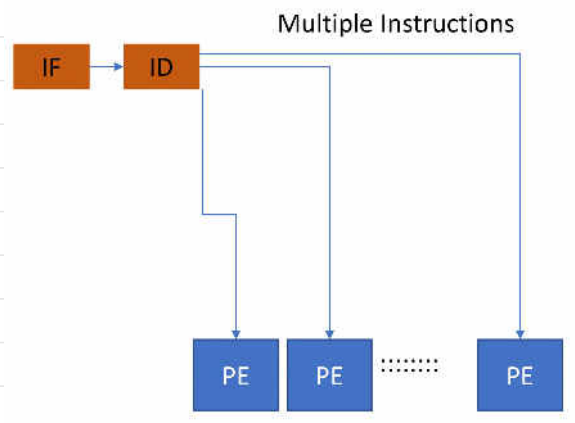
VLIW: Very Long Instruction Word

- Multiple independent instructions bundled together as a single long instruction
- Done by compiler

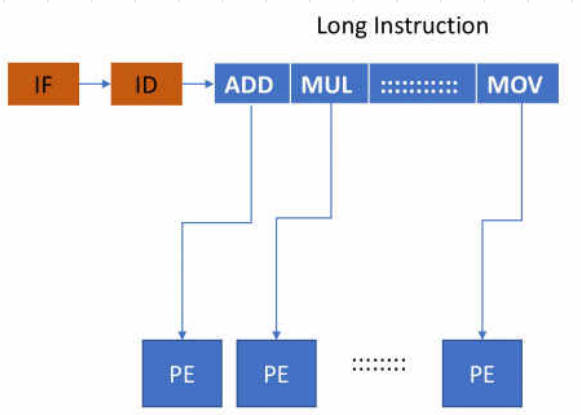
- Different parts in parallel

VLIW vs Superscalar

- Superscalar: each instruction fetched (multiple fetch) and executed in parallel



- VLIW: decode single instruction into multiple instructions and execute parallelly
- Compiler identifies independent instr & bundles



static

(dynamic more complex hardware)

Drawback of VLIW

- No independent instructions found: no-ops inserted and recompile
- Improvement: EPIC – Explicit Parallel Instruction Computer (uses speculative loading & predictions)

EPIC

- 64-bit microprocessor instruction set
- 128 general & floating point unit registers
- Speculative loading: fetch instruction and execute but do not change memory content until branch decision known
- Prediction: fetch but may not execute as branch decision is known
- Uses speculative loading, prediction and explicit parallelism

Features of EPIC

- Group of instructions – bundle
- Each bundle has stop bit: if subsequent instruction bundle depends on it
- Dependency information determined by compiler, not hardware
- Prefetching instructions: software prefetch instruction

- Check load instruction: checks whether a speculative load was dependent on a later store instruction and must be reloaded

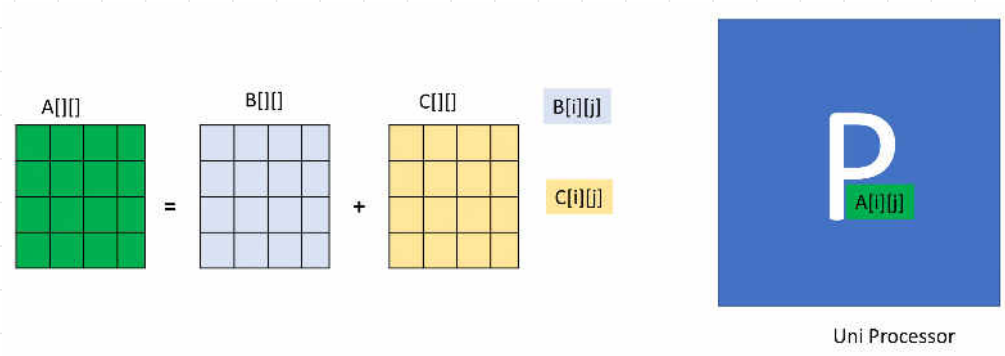
- Look at ppt references

ADVANCEMENT in PARALLEL COMPUTING

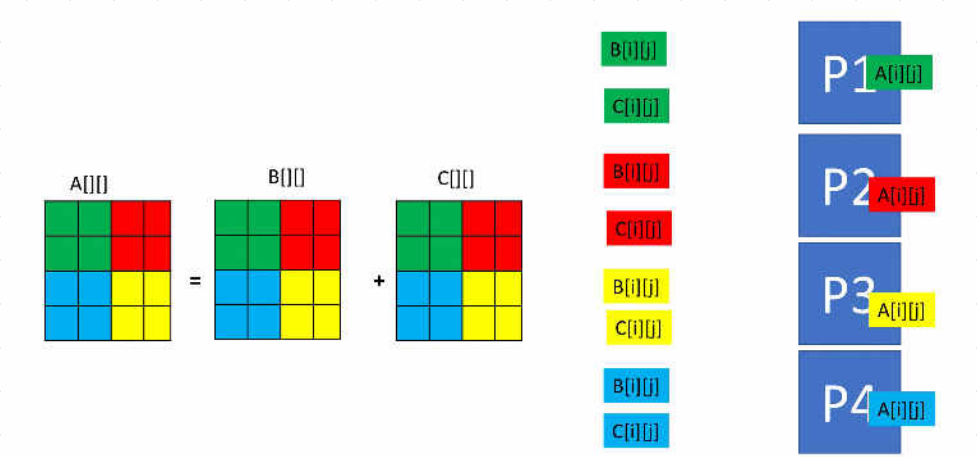
Adding Two Matrices

```
for(i=0;i<row;i++)
  for(j=0;j<col;j++)
    A[i][j]=B[i][j]+C[i][j]
```

i) Uniprocessor

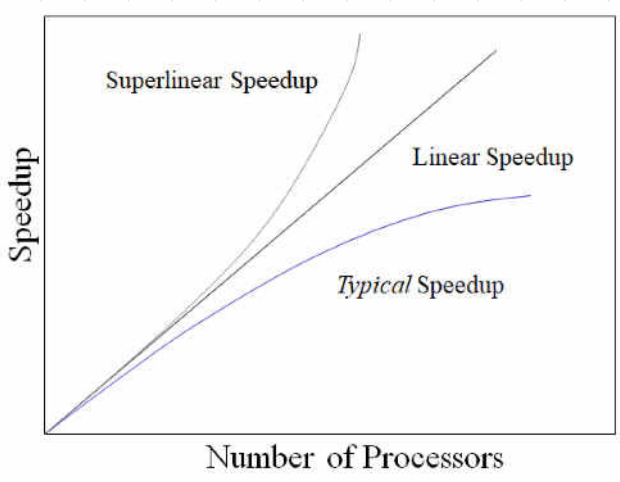


ii) Multiprocessor (4)



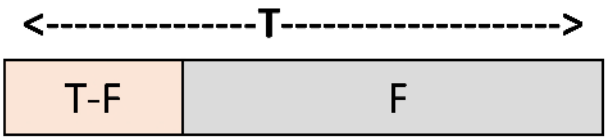
SPEEDUP

- T_s : best possible serial time
- T_n : time taken by parallel algorithm on n processors
- Speedup = $\frac{T_s}{T_n}$



Parallel Execution of a Program

- Program can be split into two parts: parallelisable and non-parallelisable parts
- T = total time of serial execution
- F = total time of parallelisable part when executed serially



- When executed parallely with n processors

$$\text{total execution time} = (T-F) + \frac{F}{n}$$

Q: The total time to execute a program is set to 1. The parallelizable part of the programs consumes 60% of the execution time. What is the execution time of the program when executed on 2 processors?

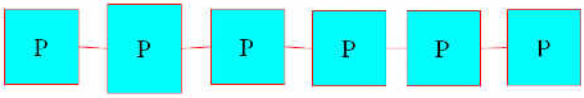
$T = 1 \qquad F = 0.6 \qquad N = 2$

$$\text{execution time} = (1-0.6) + \frac{0.6}{2} = 0.4 + 0.3$$

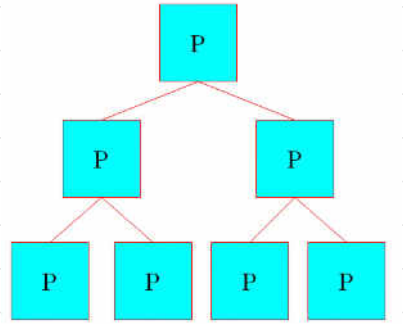
$$= 0.7$$

Parallel Computing – Processor Topology

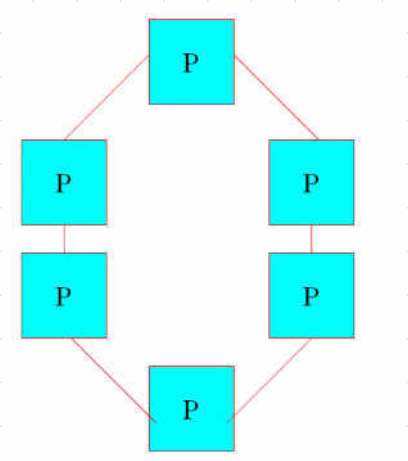
1. Linear



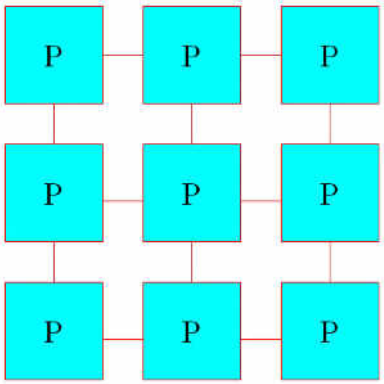
2. Tree



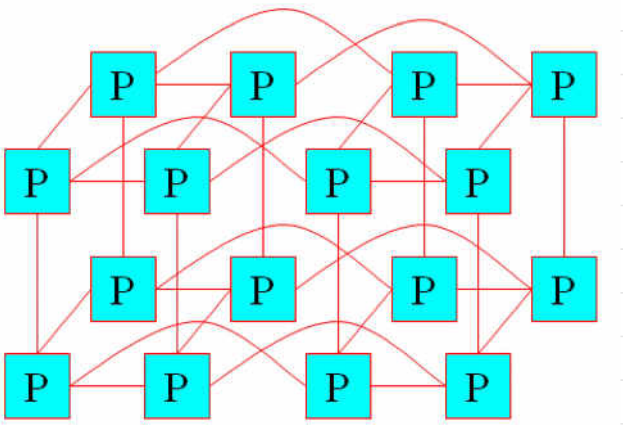
3. Ring



4. Mesh



5. Hypercube



- Look at slides for hypercube & mesh examples

AMDAHL'S LAW

$$\text{speedup} = \frac{T_S}{T_N} = \frac{T_S}{(T_S - F) + \frac{F}{N}} = \frac{T_S}{(1-f)T_S + \frac{f}{N}T_S}$$

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{n}}$$

if n = speedup factor s, more generally

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{s}}$$

s = speedup factor

f = fraction of program that can be optimised

$1-f$ = fraction of program that cannot be optimised

Limit : as $s \to \infty$

$$\text{max speedup}(f) = \frac{1}{1-f}$$

Q: What is the overall speed up if 10% of the program is made 90 times faster?

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{S}}$$

$f = 10\%$
$S = 90$

$$\text{speedup} = \frac{1}{0.9 + \frac{0.1}{90}} = \frac{900}{811}$$

$$= 1.11$$

Q: What is the overall speed up if 90% of the program is made 10 times faster?

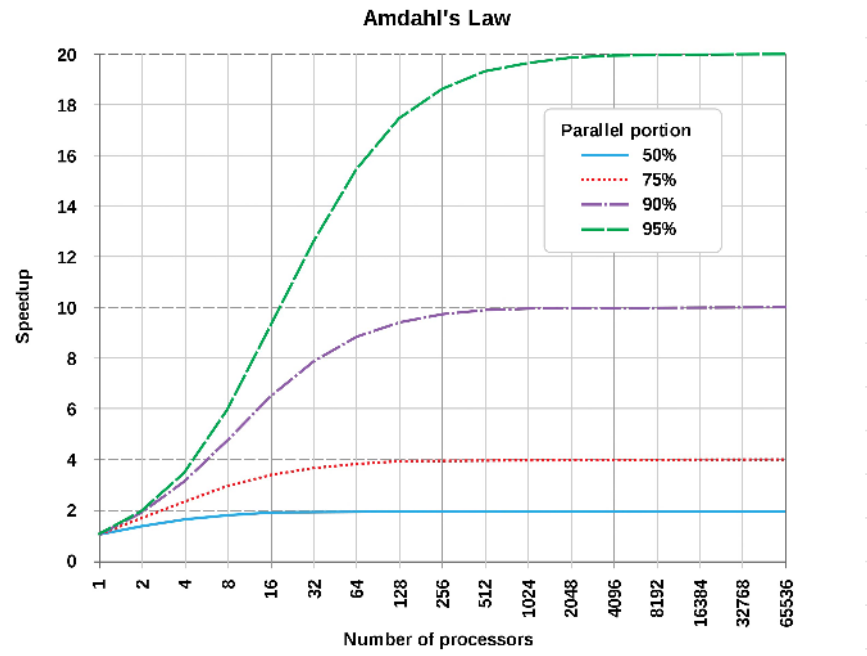
$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{S}}$$

$f = 0.9$
$S = 10$

$$\text{speedup} = \frac{1}{0.1 + \frac{0.9}{10}} = \frac{100}{19}$$

$$= 5.26$$

$$\text{max speedup}(f) = \frac{1}{1-f}$$



source: Wikipedia

(i) $f = 0.5 \Rightarrow \text{speedup}_{max} = 2$

(ii) $f = 0.75 \Rightarrow \text{speedup}_{max} = 4$

(iii) $f = 0.90 \Rightarrow \text{speedup}_{max} = 10$

(iv) $f = 0.95 \Rightarrow \text{speedup}_{max} = 20$

Flaws with Amdahl's Law

- Assumes speedup independent of problem size
- Does not account for scalability
- Ignores communication cost

GUSTAFSON'S LAW

- The proportion of sequential computations decreases as the problem size increases
- Not theorem; observation
- Assume parallel execution time fixed

$$\text{speedup factor} = N - (N-1) * S$$

S = serial part of code (fraction)

- Increase no. of processors as well as program size

Q: Suppose a program has serial section of 5% and 20 processors. Find speedup according to Amdahl's and Gustafson's laws.

$$f = 0.95$$

(i) Amdahl's law

$$\text{speedup} = \frac{1}{0.05 + \frac{0.95}{20}} = \frac{400}{39} = 10.26$$

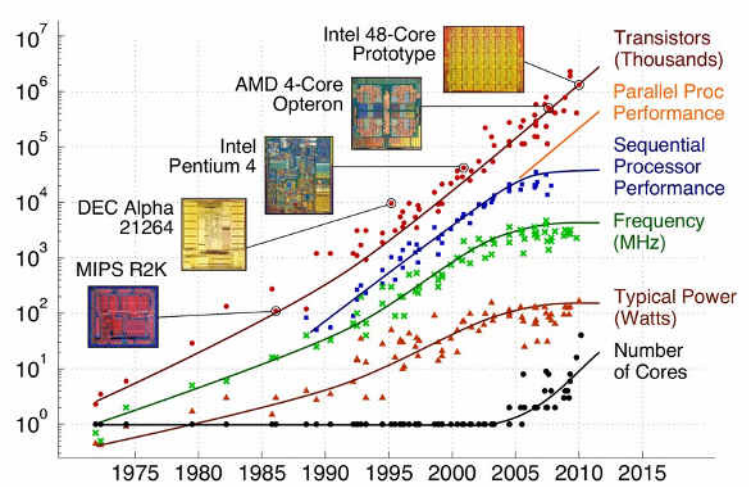
(ii) Gustafson's law

$$\text{speedup} = 20 - (19) \times 0.05$$

$$= 19.05$$

MULTICORE PROCESSORS

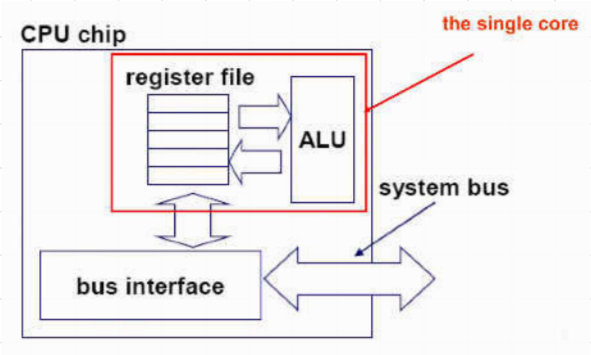
- Frequency limit ; parallelisation required



Limitations of single core

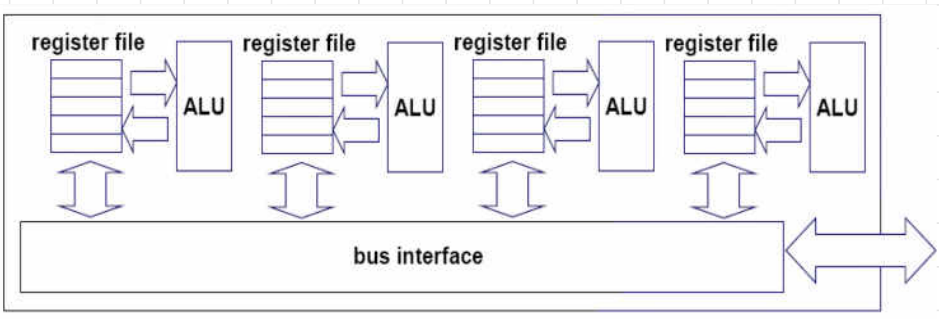
- Power wall (heat)
- Memory wall (mem access latency)
- ILP — instruction level parallelism — wall (dependency, instruction window size for fetch)

Single Core CPU CHIP



- Can be considered as one thread
- Single-threaded processor
- Register, PC, SP, interrupt logic, execution unit, cache

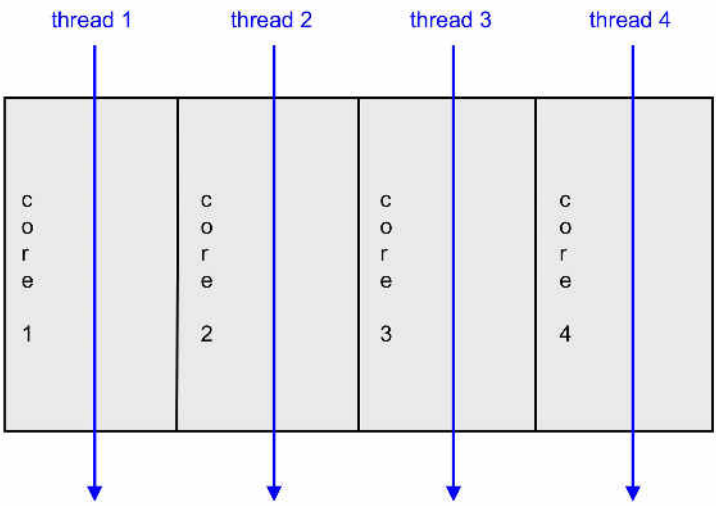
Multi-Core CPU CHIP



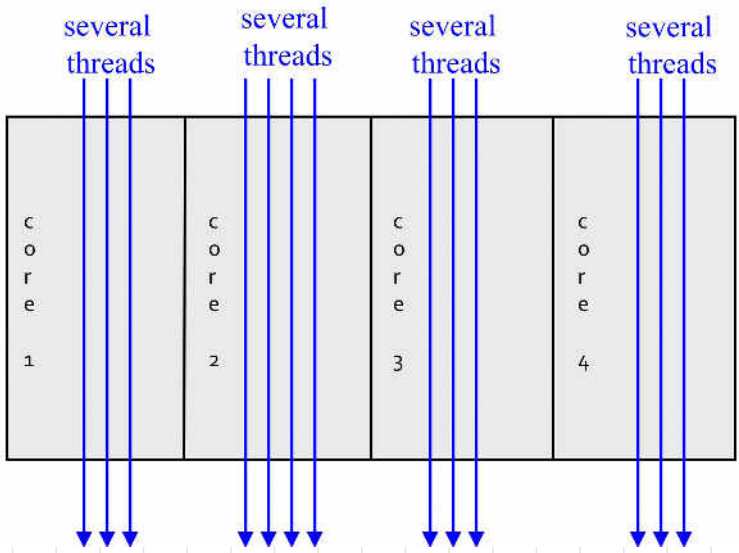
- Chip multiprocessing (CCMP)
- Multicore with hyperthreading
- MIMD — different cores execute different threads operating on different parts of memory
- Shared memory multiprocessor

Multi-Core Architecture

- One thread in each core



- Several threads in each core



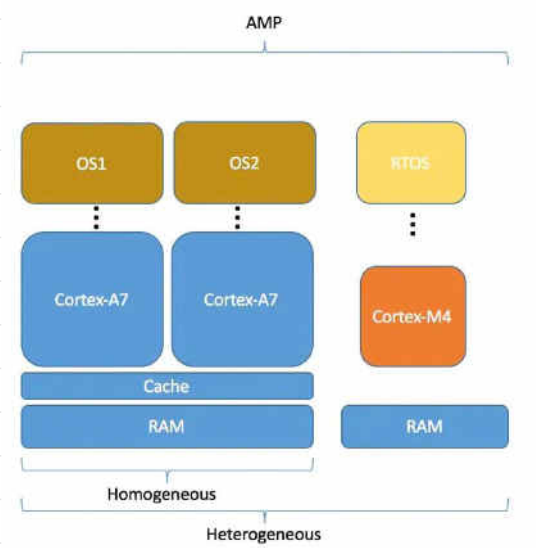
AMDAHL'S LAW FOR MULTICORE PROCESSORS

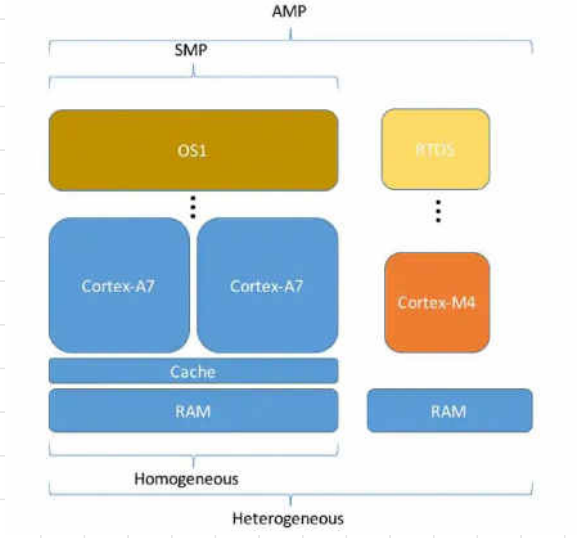
$$\text{Speedup} = \frac{1}{\frac{1-F}{\text{Perf}(R)} + \frac{F*R}{\text{Perf}(R)*N}}$$

not coming for exam

Homogeneous & Heterogeneous Multiple Core Architecture

- Identical processor cores: same instruction set architecture
- Non-identical processor cores: different ISAs
- Eg: ISAs for GPU & CPU processors



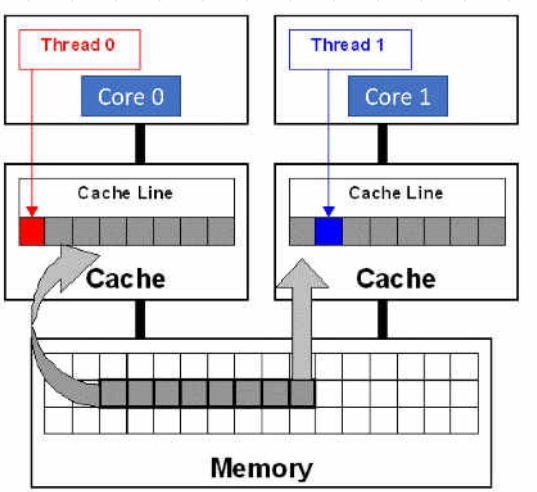


Roles

- User
  - use threads / processes
  - spread workload
  - write parallel algorithms

- OS
  - maps threads to cores
  - each core perceived as processor
  - major OSes support multicore

- Memory
  - memory contention (bandwith shared for communication and computation)
  - memory refers to cache
  - cache coherence protocols
  - RAM on chip: M1 (Unified Memory Architecture - UMA)
  - False Sharing: shared cache; if multiple processors accessing same cache line / block to write, lots of unnecessary bus traffic for cache coherence (CPU cache line interference)



Players

- User
- OS
- Compiler
- Hardware

} tasks distributed