Course Outline

. General Overview – HPC
. Hardware + basic optimization techniques
. Parallel algorithm design
. Shared and Distributed
. Technical (OpenMP and MPI)
. Performance modeling of parallel algorithms
. Important parallel patterns

. Application (assignments and project)

Lab 207

Operating system : Scientific Linux
Compilers and Libraries: The entire GNU compiler suite - gcc, g++, and g77
The Java Execution and Development Environments.
Python programming language. Also Matplotlib, Numpy and Scipy

Parallel Programming Libraries and tools:

OpenMP - API for directing multi-threaded shared memory parallelism.
OPENMPI- open source implementation of MPI (access to HPC Cluster)
GNU Gprof - performance analysis tool for Unix applications.

Important Scientific Software: Scilab; Octave; R - software environment for statistical computing and graphics.

Visualization: Gnuplot. Paraview.

Documentation and reader: Latex

Access to HPC Cluster

Module 1: Main topics

1. Introduction to high performance computing and Systems Perspective.
2. Thinking in serial vs thinking in parallel.Need for Parallel Computing.
3. Performance Metrics : Compute performence and memory performance. Latency, bandwith, throughput. Balance Analysis.
4. Modern Processors
5. Pipeline and Memory Hierarchies
6. Data access optimizations. Performance Modeling- benchmarks: Vetor Triad. Measuign performance and profiling.
7. Dependences and loops
8. Scope of parallelism and Parallel computing metrics

References:

** Introduction to HPC for Scientists and Engineers by G Hager and G Wellein*
** Let's HPC: A web-based platform to aid parallel, distributed and high performance computing education*
https://doi.org/10.1016/j.jpdc.2018.03.001

Goal of HPC: to achieve the maximum possible performance out of a particular system for a particular problem

- Performance → Time (Work/time → Rate)
- System → Architecture (Computing) → Non-deterministic in nature
- Problem → Algorithm (Application)

Need to understand as a whole not in parts.

What is expected from a good programmer?

- Accuracy → Correctness
- Efficiency (performance - time)

Accuracy is proportional to no of Computations

Evaluation of the program in the given software-hardware setting

Algorithm design

Thinking in serial

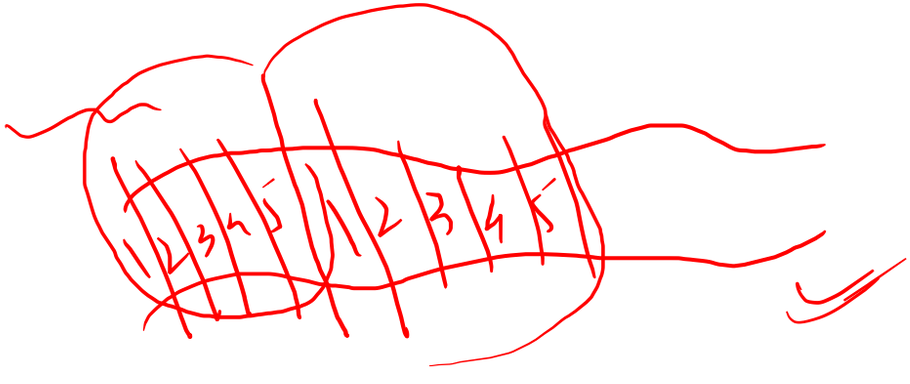
Resource → Searching tool

5 hrs

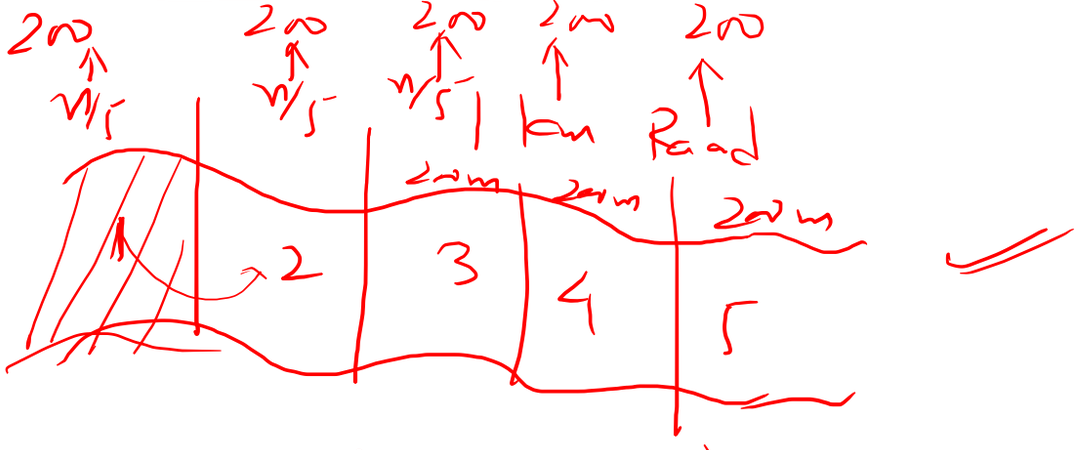
Adding n numbers

for i = 1 n

end



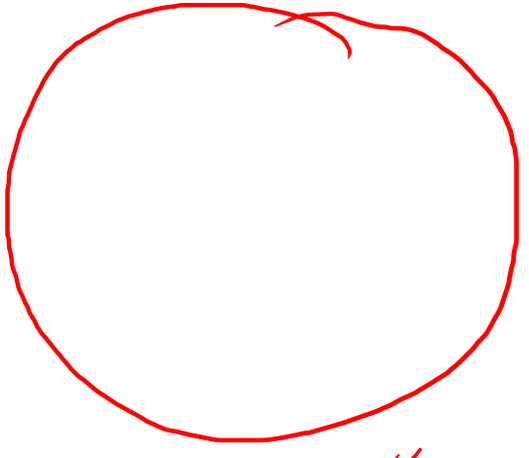
Thinking in Parallel

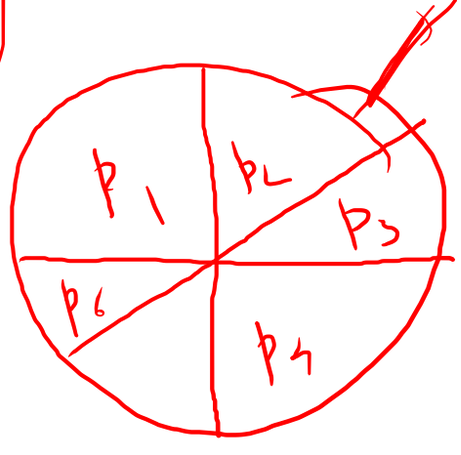


* Problem decomposition (ideal) 5 person
* Resource for every person → 1 hr

~~for i = 1 200~~ | for i = 201, 400 | for i = 401, 600 | for i = 601, 800 | for i = 801, 1000

partial Avg | partial Avg | partial Avg | partial Avg | partial Avg





Dig a hole

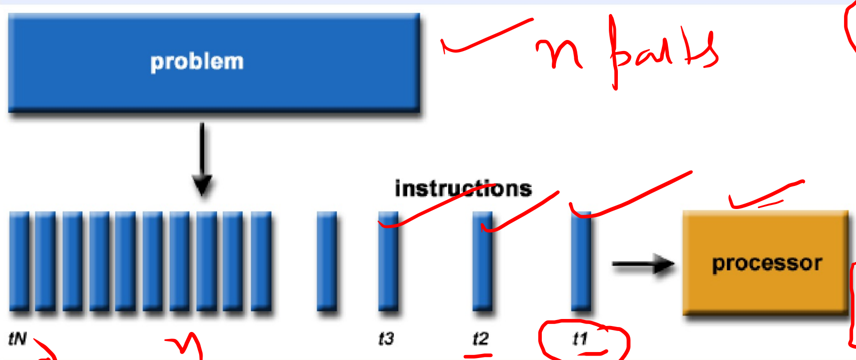
1m diameter
&
1m deep.

~~5~~ philosophers

Philosophy

10 processors

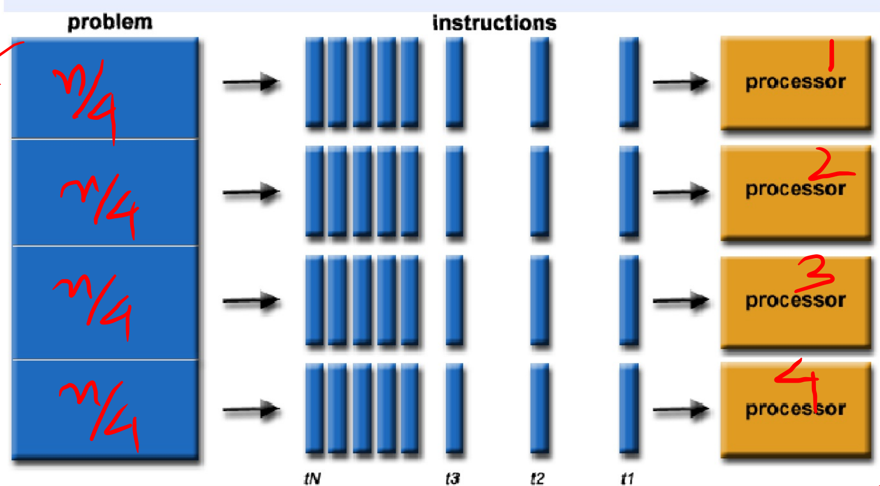
Serial vs Parallel



Serial

- Break problem → discrete series of instructions
- Instructions executed → sequentially.
- Executed on a single processor
- Only one instruction @ any moment in time .

Improve Latency.



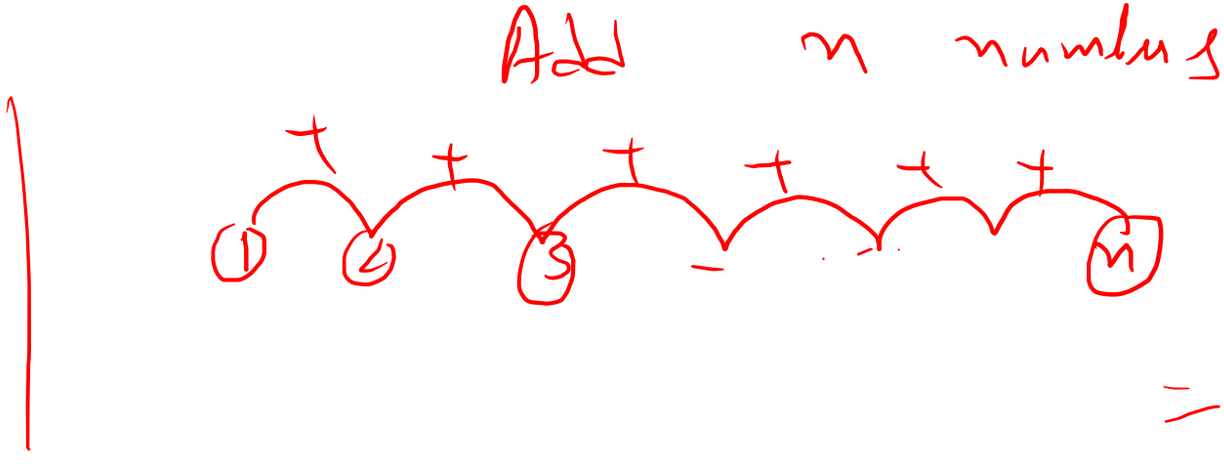
Parallel: multiple compute resources.

- Break problem → discrete parts that can be solved concurrently.
- Each part again broken down to a series of instructions
- Each instructions (from each part) → execute simultaneously on several processors
- An overall control/coordination mechanism is needed.

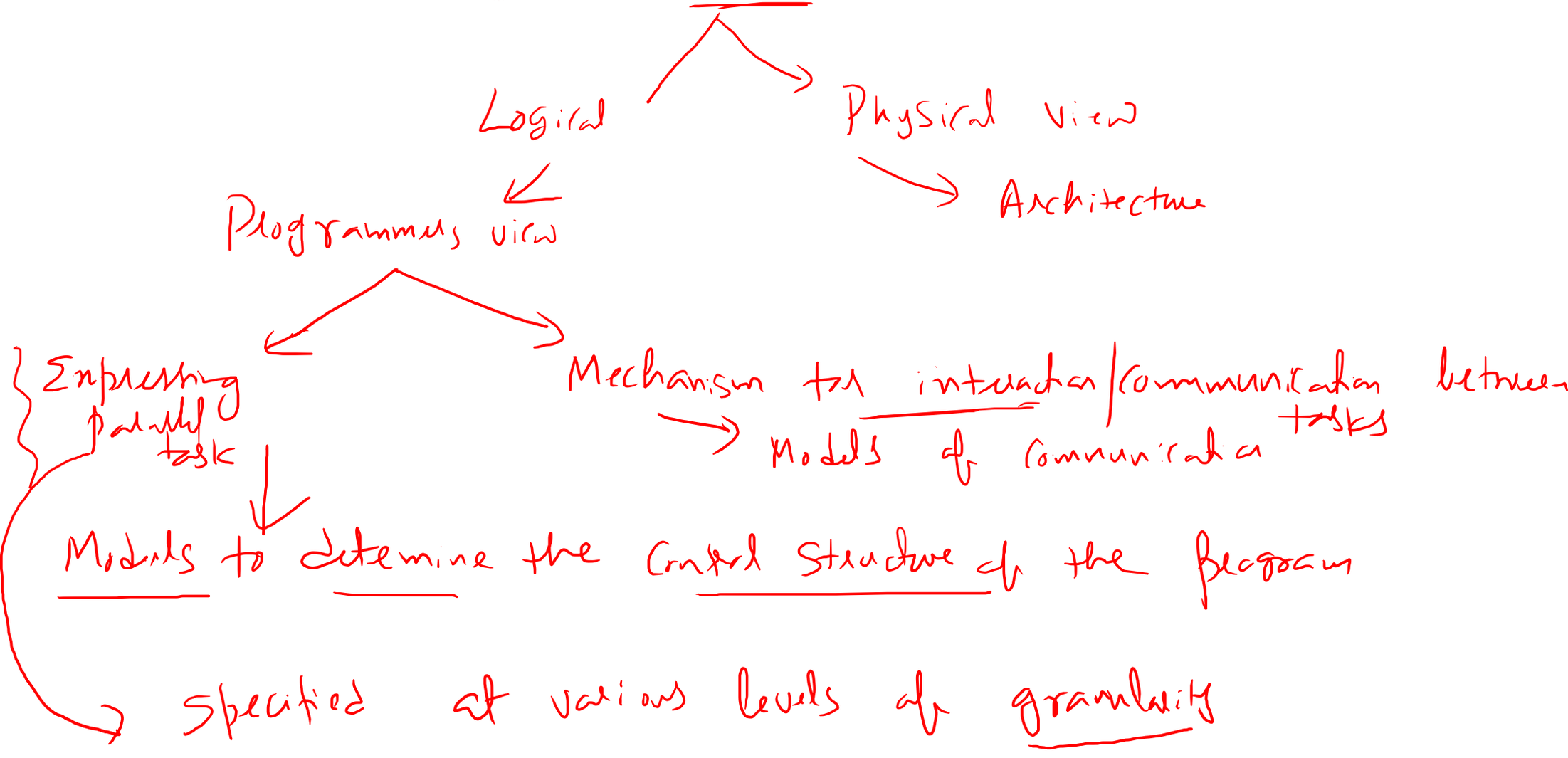
Improve throughput.

We need a systems perspective : why?

systems are in general non-terminating and non-deterministic, whereas the behavior of algorithms is terminating, deterministic and platform- independent



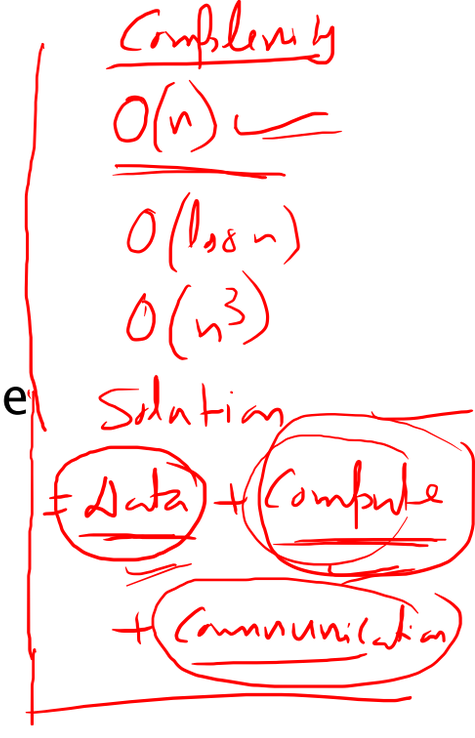
Organization of Parallel Platforms

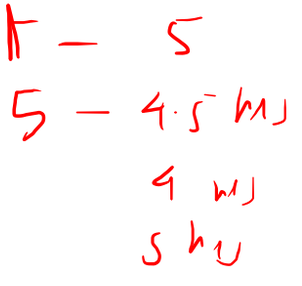


Theory of parallel algorithm design or theory of parallel computing is based on abstract concepts of time and memory which may ignore real life constraints for simplicity, and therefore not take into account non-deterministic and hardware factors

The performance of a computer program depends on a wide range of factors like the nature of the algorithm, the machine (several hardware factors), compiler optimizations, the runtime environment, the input, the measurement methodology etc. and their mutual interaction

Not enough just to get some speedup, but being able to explain the speedup from a systems viewpoint in terms of resources.





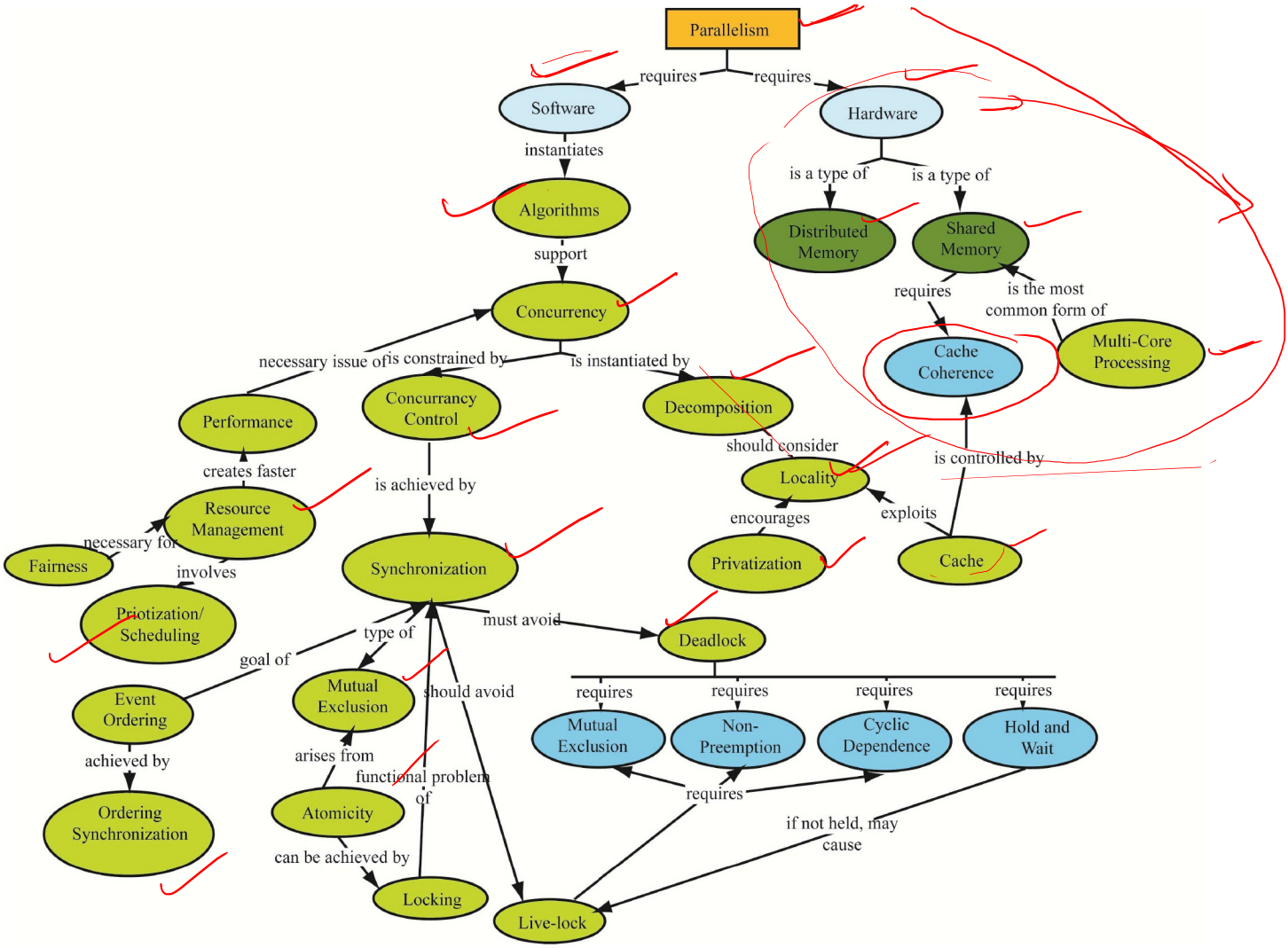


Figure 1. Concept map for parallel computing. Adapted from [9].

Course content

Module 1:

Modern Processor, Performance Metrics and benchmark, Memory Hierarchies, Caches, Pipelining, SIMD, SPMD, performance estimates, profiling, vector triad, serial code optimizations, data access optimizations, balance analysis, dependence testing, granularity, concurrency, loop fusion, loop distribution. Cache miss, cache hit, block matrix multiplication. Amdahl's law.

Computer Performance

What is HPC and goal of HPC ??

Quantitative measure !!

Computer Performance

What is HPC??

- Measure of computer performance -> **FLOPS** (**FL**oating-point **O**perations **P**er **S**econd).
- Performance → We need Clock speed and microprocessor (Flops/cycle)
- FLOPS= **clock** rate x (flop/cycle) x **no. of cores**
- Microprocessors today can do 4 FLOPs per clock cycle.
- A single-core 2.5 GHz processor has a theoretical performance of 10 e9 FLOPS = 10 GFLOPS.

Aggregating computing power in such a way that delivers much higher performance than a typical desktop computer (Look at FLOPS formula !!.) -→ **to solve large problems.**

Useful in - science, engineering, or business.

Lecture 2
CS301 - HPC
Course Instructor : B. Chaudhury

Previous Lecture

Performance= Work/Time = FLOP/Wall clock time

FLOP (addition, multiplication, division etc.)

FLOPS --> Number of FLOP/Second

FLOPS (flops per second) can be used to characterize a computing system (peak theoretical performance) and it can also be used to characterize a program (code) i.e. actual performance of the program in terms of how many flops per seconds

High performance computing trend

1 exaFLOPS (EFLOPS)
Before 2020.

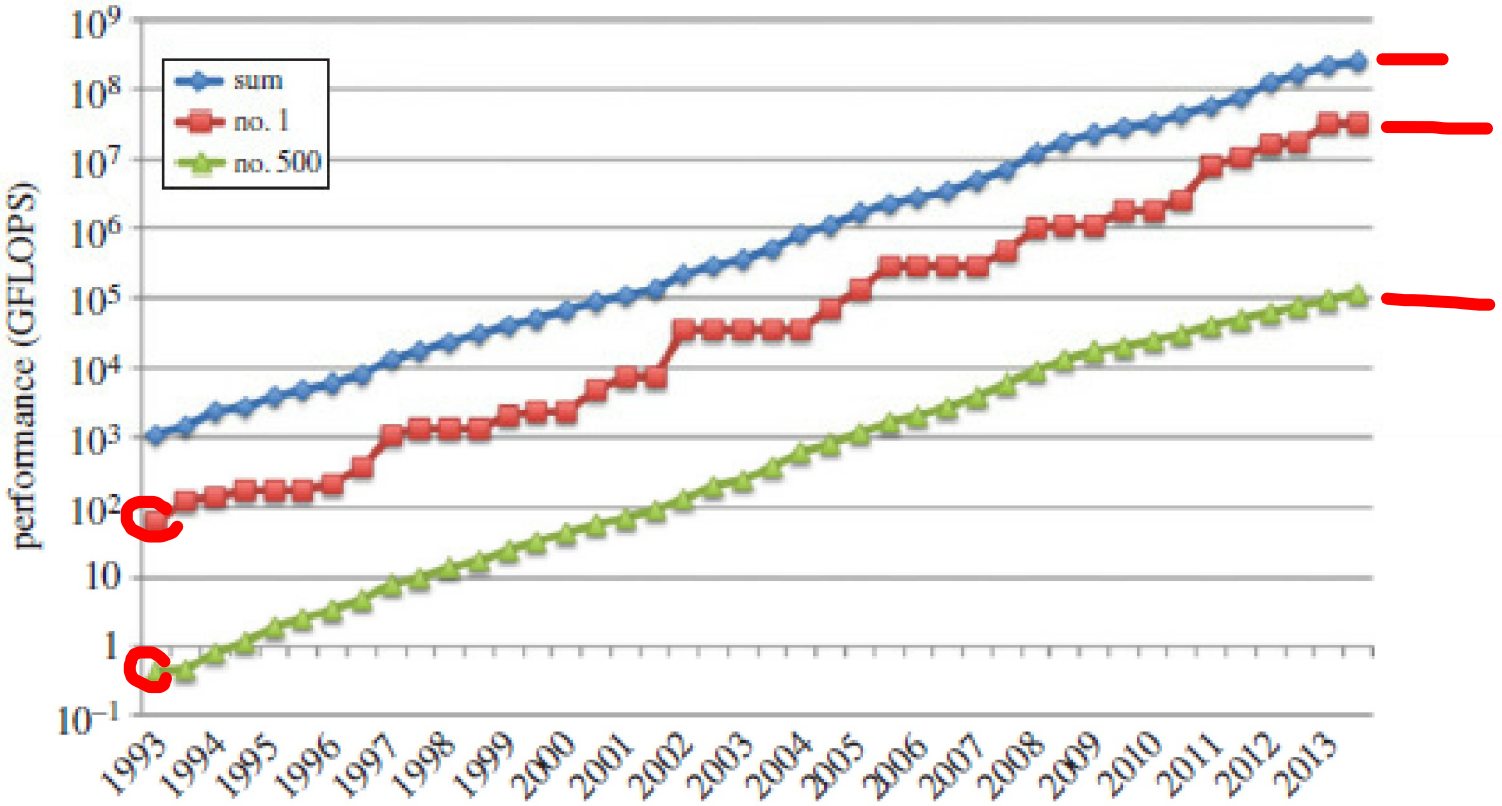


Figure 1. Data showing performance of Top500 supercomputers on the LINPACK test over the past 20 years. The top line is the sum of the top 500 systems, the middle line is no. 1, and the bottom line is no. 500 [1]. (Online version in colour.)

http://dx.doi.org/10.1098/rsta.2013.0319

HPC trends

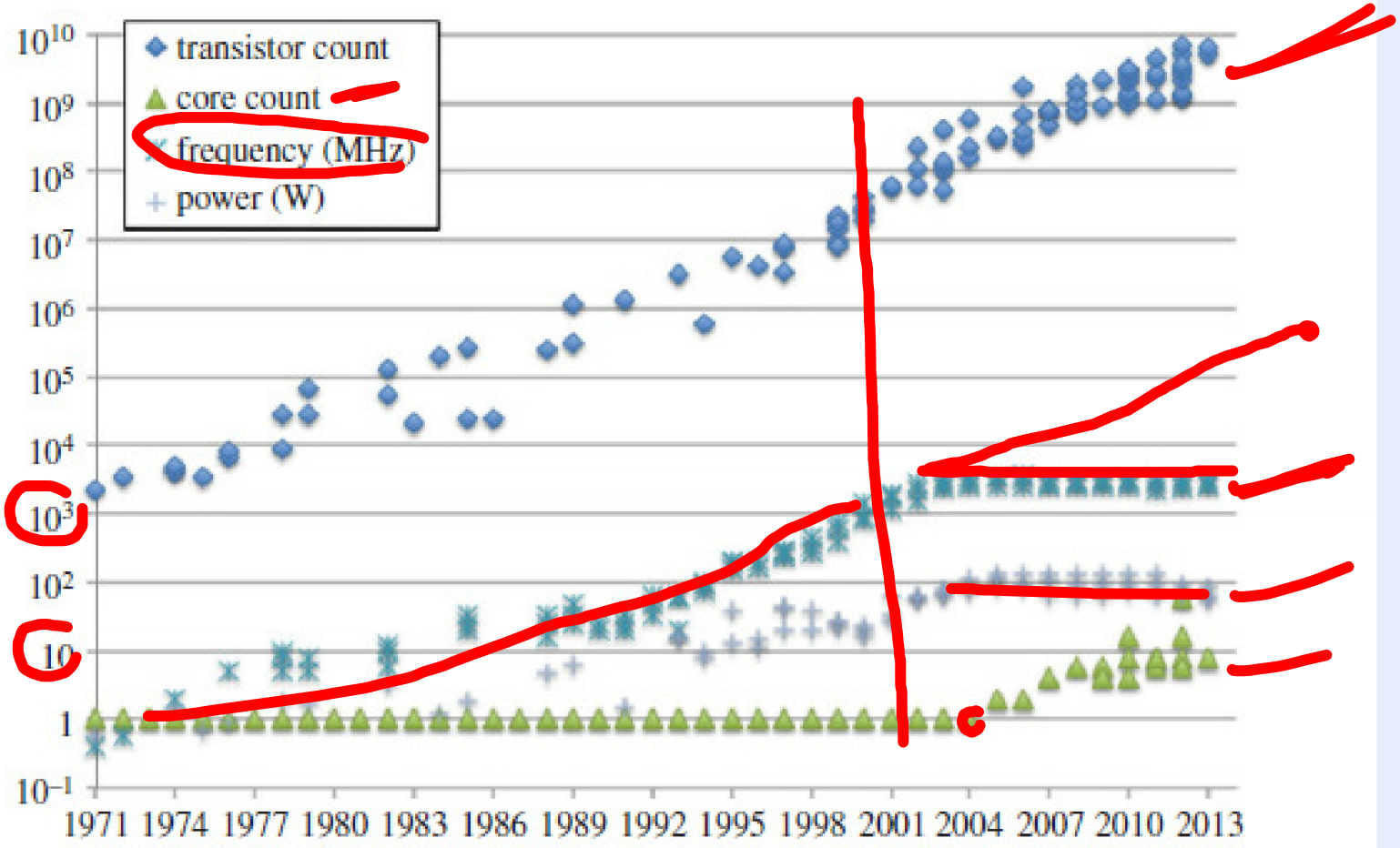


Figure 3. Evolution in transistor count, clock frequency, number of cores and power consumption for processors over the past 40 years. (Online version in colour.)

- No more serial --- It's a Parallel World.....We need to know how to work with these instruments!

HPC Applications

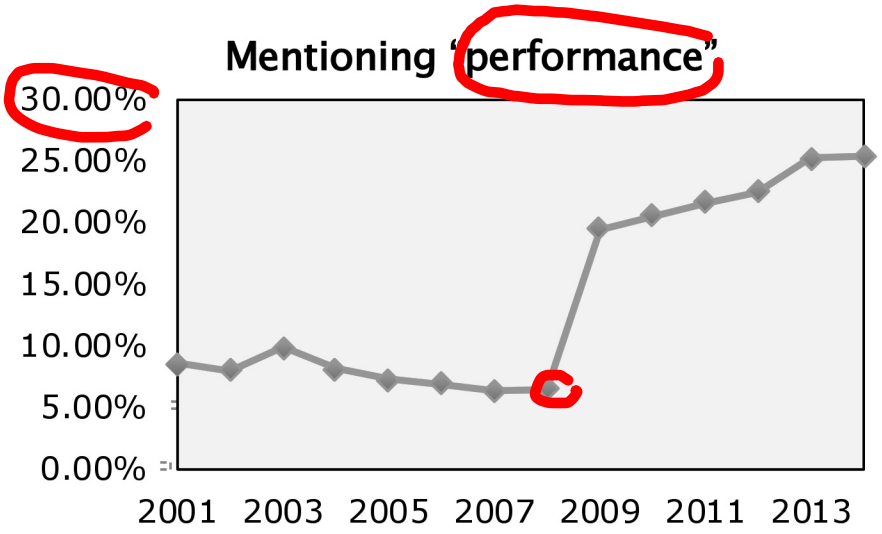
Science and Engineering:

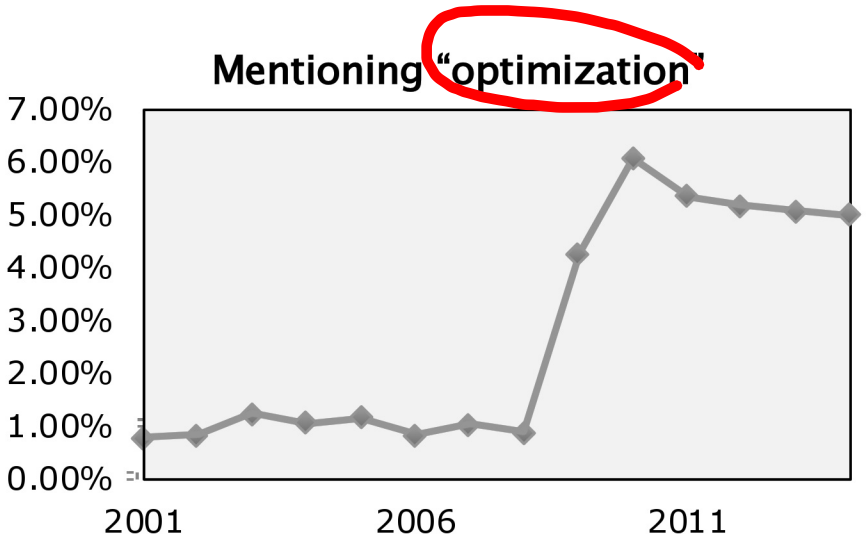
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion.
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Mechanical Engineering
- Electrical Engineering, Microelectronics
- Computer Science, Mathematics
- Defense, Weapons

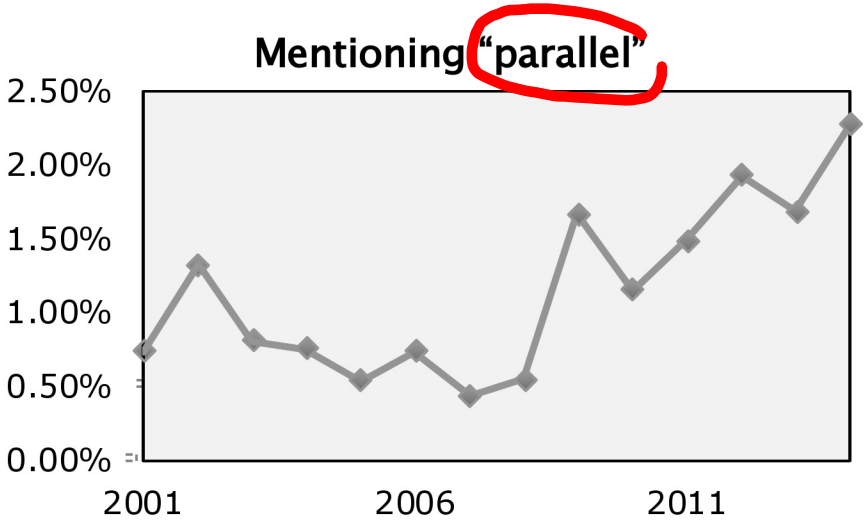
Industrial and Commercial:

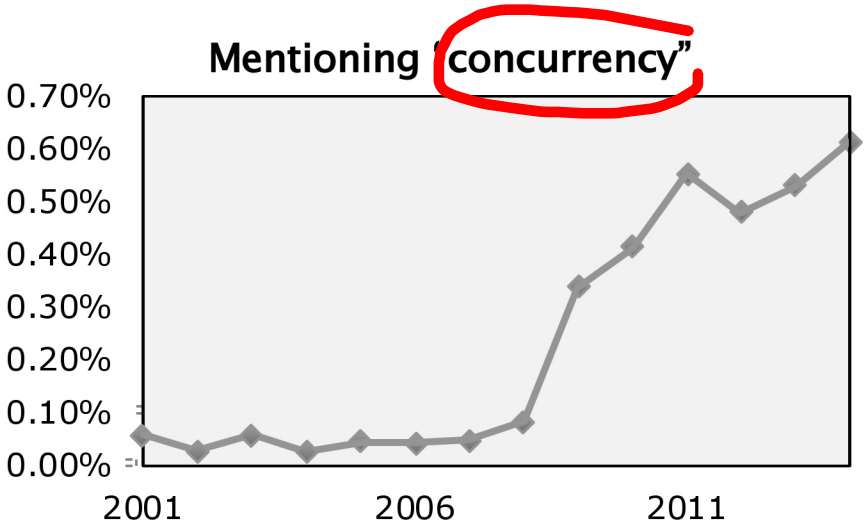
- Databases, data mining
- Oil exploration
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial modeling
- Entertainment industry
- Web search engines, web based business services

Software Developer Jobs









Source: Monster.com

What do we learn?

~~Serial~~ Performance Scaling is Over

! Cannot continue to scale processor frequencies
! no 10 GHz chips

! Cannot continue to increase power consumption
! can't melt chip

! Can continue to increase transistor density
! as per Moore's Law

You *must* re-think your algorithms to be parallel !

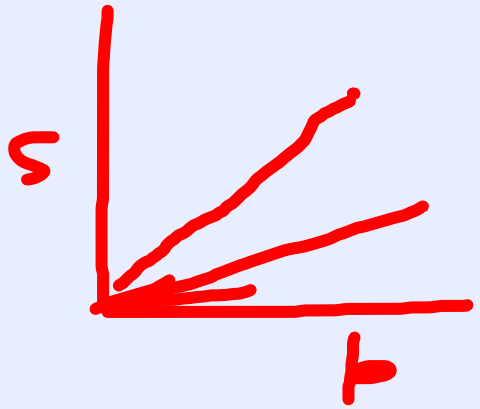
High Time to learn Parallel Programming →

- Why parallel computing ?
- What should parallel computing achieve ?
- Which disciplines parallel computing should address ?

Conventional Wisdom	Conceptual shift
Power is Free, Transistors are expensive	Transistors are free, Power is expensive → Power Wall
Uniprocessor performance doubles every 18 months (Moore's law)	Power Wall + Memory Wall + ILP Wall → Brick Wall. Now doubling uniprocessor performance may take 5 years as per recent trend
Hardware design can increase in size with more transistors without any negative impact	Wire delay, clock jitter, noise, cross coupling stretches the limits in design for feature size < 65nm
Multiply functional unit is slow but load and store memory operations are fast	Due to increasing DRAM gap, load and store are much slower then floating point multiplication → Memory Wall
Instruction Level Parallelism (ILP) through Out-of-Order execution, speculation, compiler optimization such as loop unrolling improves performance of "multicore" architectures	Diminishing results for ILP in "manycore" architectures → ILP Wall

continued

Conventional Wisdom	Conceptual shift
Increase in frequency is primary method of improving processor performance	Cant burn the processor by increase in frequency. Increasing parallelism is primary method of improving performance
Less than linear scaling in performance for multiprocessor application is a failure	By switching to parallel computing, any speedup via parallelism is a success



Applications

Application Domain	Application Examples
Embedded Computing	Consumer: JPEG, RGB to CYMK, Entertainment: MPEG decode-encode, Networking: IP NAT, OSPF, Route Lookup, Automation: Text Processing etc
General Purpose Computing	Computational science, Fluid dynamics, Molecular dynamics, Computational Electromagnetics, Weather modelling, network simulation, XML transformation , Video compression, etc
Machine Learning	Support Vector Machines, Principal Component Analysis, Spectral Clustering, Expectation Maximization, Bayesian Networks etc
Graphics and Game	Reverse Kinetics, Spring models, Texture Maps, Smoothing, Interpolation, Collision Detections and Responses etc
Databases	Query Optimization, MapReduce, Hashing etc

Common kernels

Numerical Algorithm behind an Application

ID	Dwarf	Description
1	Dense Linear Algebra	Dense Vector-Vector, Matrix-Vector, Matrix-Matrix operations, Use of stride access for rows and columns
2	Sparse Linear Algebra (SpMV etc)	Sparse data (with many zeroes) stored in compressed format. Data access with indexed load and store
3	Spectral Methods (FFT etc)	Data in frequency domain, data accessed in multiple butterfly stages with all-to-all for some stages and local to others
4	N-Body methods	Depends on interaction between many discrete points
5	Structure Grids	Data in regular grid format, points on grid updated together with high spatial locality. Subdivides grids into finer grids with area of interest
6	Unstructured Grids	Data locations in grids as per application characteristics

Common kernels

First 7 kernels/dwarfs covers most computational science applications

7	**Monte Carlo or MapReduce**	**Calculation is done as per repeated random trials which are embarrassingly parallel**
8	Combinational Logic	Implemented using logical function and stored states used to perform simple operation on large amount of data e.g. calculating CRC
9	Graph Traversal	Visit nodes following successive edges, computationally less expensive
10	Dynamic Programming	Solve simpler overlapping problems, useful for optimization
11	Backtrack and Branch+Bound	Finds an optimal solution by recursively dividing the feasible region into subdomains, and then pruning sub-problems that are suboptimal
12	Graphical Models	Graphs with nodes as random variables and edges as conditional dependencies. E.g. Bayesian Network, Hidden Markov Models
13	Finite State Machine	System behavior defined by states, transitions defined by input and current state and event associated with transitions or states

Important Patterns

Dwarf/Kernel	Embedded Computing	General Computing	Machine Learning	Graphics and Games	Databases
Dense Linear Algebra	√	√	√		√
Sparse Linear Algebra	√	√	√	√	
Spectral Methods	√		√	√	
N-Body methods		√			
Structure Grids	√	√		√	
Unstructured Grids			√		
Monte Carlo (MapReduce)		√	√		√

Common patterns

Dwarf/Kernel	Embedded Computing	General Computing	Machine Learning	Graphics and Games	Databases
Combinational Logic	√		√		√
Graph Traversal	√		√	√	√
Dynamic Programming	√	√	√		√
Backtrack and Branch+Bound		√	√		
Construct Graphical Models	√	√	√		
Finite State Machine	√	√		√	

Challenges

- Application design may have combination of different kernels → How to use architecture and programming model tuned in individual kernels/dwarfs for combination design
- Algorithm implementation for different kernels have preferred data structure → Solution having combination of different dwarfs require data structure translation

Terminology

- The cores perform FLOPS.
- A processor contains one or more (CPU) cores.
- A socket holds one processor.
- A node contains one or more sockets.

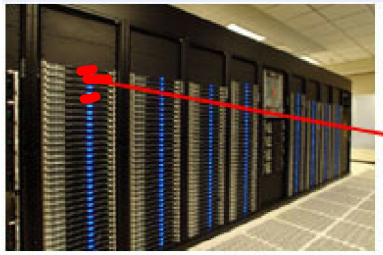
Supercomputer = several nodes → HPC

HPC world → node peak theoretical performance:
Node performance in GFlops = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node).

CPU → Serial & Parallel (MPI, OpenMP etc.)

How many cores we need for 1 Teraflop (cores@2.5 GHz)?

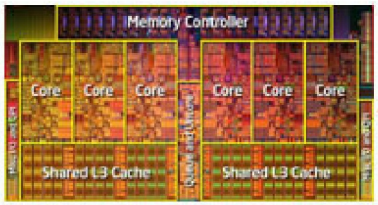
Terminology --- visual

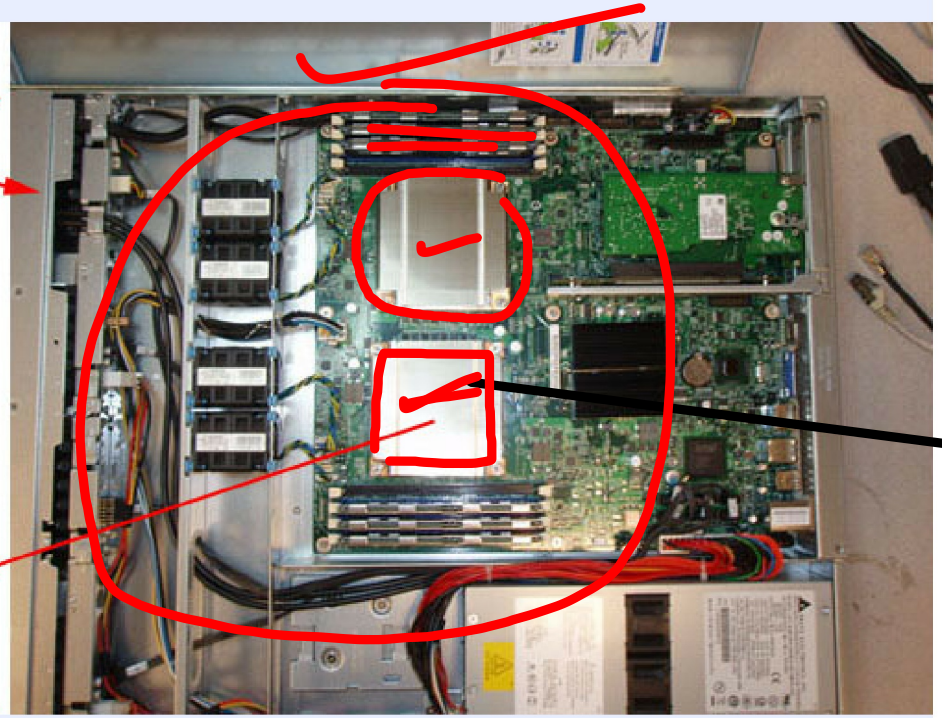


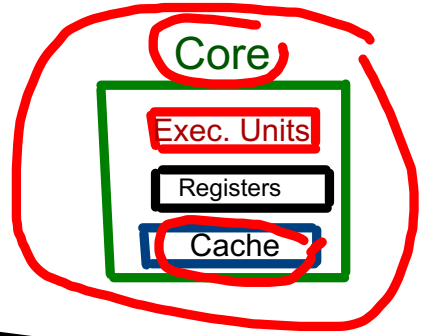
Supercomputer - each blue light is a node

Single O/S — Node - standalone Von Neumann computer

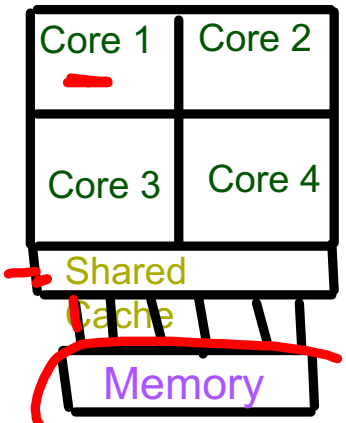
CPU / Processor / Socket - each has multiple cores / processors.







Processor/Socket



The same Parallel code can be executed on a laptop (multi-core) and can be also executed on a supercomputer

What do we learn?

Serial Performance Scaling is Over

! Cannot continue to scale processor frequencies
! no 10 GHz chips

! Cannot continue to increase power consumption
! can't melt chip

! Can continue to increase transistor density
! as per Moore's Law

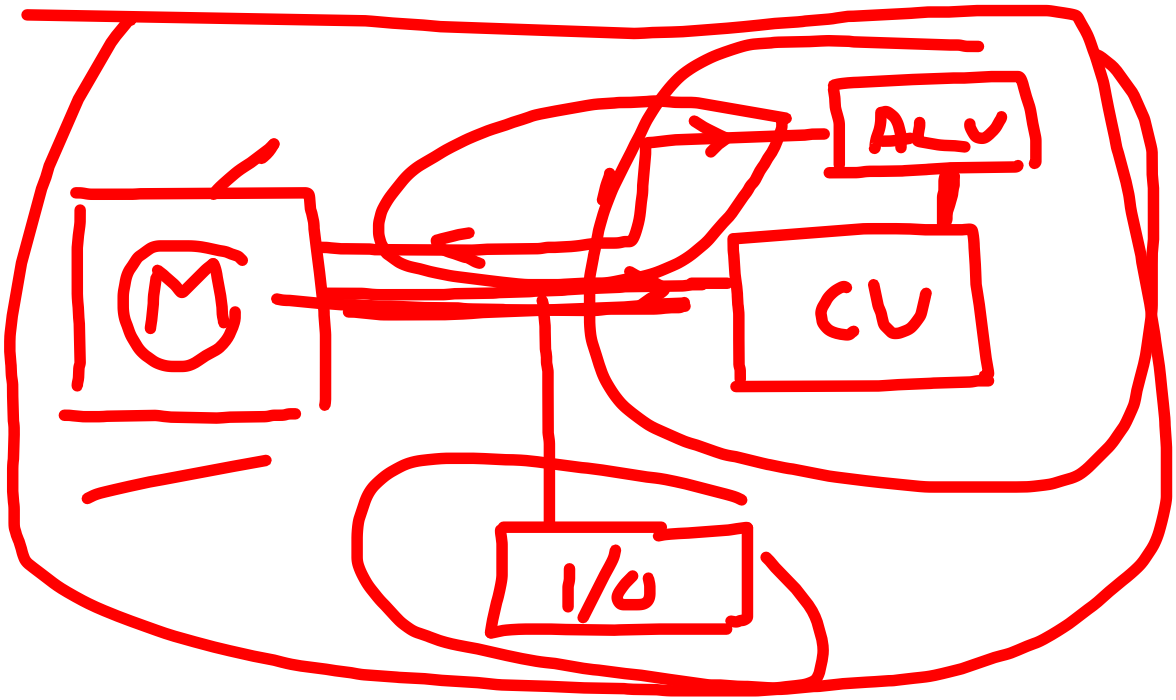
You ***must*** re-think your algorithms to be parallel !

High Time to learn Parallel Programming →

Modern Processor and Principle of multiplicity

Single core Architeture (stored program digital computer)

Instructions are stored as data in memory



CU - instructions read & execute
ALU - computation
Memory - store data (for ALU) and instructions (for CU)
I/O - communication with user

Different ways to classify parallel computers

* Instruction Stream
* Data Stream

Two possible states
(single or multiple)

Stream refers to a sequence or flow of either instruction or data operated on by the computer

* Instruction Stream: flow of instructions from main memory to CPU
* Data Stream : flow of operands between processor and memory (bi-directional)

Performance From where?

- Device Technology
  - *Logic switching speed*
  - *device density*
  - *Memory capacity and access time*
  - *Communications bandwidth and latency*
- Computer Architecture
  - Execution pipelining
  - Cache management
  - *Parallelism*
    - Parallelism – number of operations per cycle per processor
      - Instruction level parallelism (ILP)
      - Vector processing
    - Parallelism – number of processors per node
    - Parallelism – number of nodes in a system

```
Do i = 1, N
   A(i) = B(i) + C(i)
```

$N = 10^6$

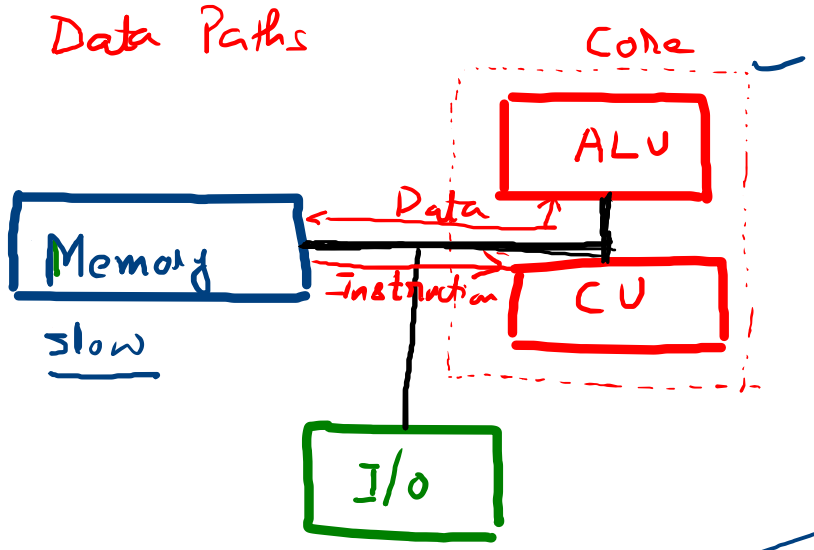
$Perf = \frac{10^6}{Time(s)}$

Instruction →

Runtime → Data transfer + Calculate + Misc

Lecture 3
CS301 - HPC
Course Instructor : B. Chaudhury

Previous Lecture



Main performance bottleneck
- Memory Access is slow

Performance depends on (Runtime)
- Instruction Execution
- Data Movement

```
Do i=1,N
A[i] = B[i] + C[i]
end
```

① Total work → N adds → N FLOPS ← Programmer.
↳ Several Instructions (load, store, Exec.) ← Architect

② Data transfer → How much ?
8 bytes each A, B, C
24 bytes per iteration

Measure ?
Bytes/Sec

Classification of Computers: Some Computer organizations and their effectiveness by M. J. Flynn, 1972.

Understanding the effects of Architecture

Stored Program computer.

Von Neumann architecture → main memory, CPU and interconnection.

Von Neumann bottleneck ?

Improvements → modifications to von Neumann model.

- Caching.
- Pipeline

Performance Metrices and Benchmarks

We know that all components can operate at some max speed called peak performance.

Two imporant quantities :
Compute Throughput (GFLOPS/Sec)
Memory Bandwidth (GBytes/Sec)

Low level Benchmark: Program that tries to test some specific features of the architecture.

Isolate small set of instructions to separate influences and determine specific machine capabilities

Vector Triad - popular microbenchmarking exercise
Multiply-Add nested loop on the elements of three vectors

Vector Triade A[i]=B[i] +C[i]*D[i]

```
int minSize = pow(2, 3);  int maxSize = pow(2, 29);
int total = maxSize;

for(int size=minSize; size<=maxSize; size*=2) {
    /* init data */
    for(int i=0; i<size; i++) {
        b[i]=3; c[i]=2; d[i]=1;
    }

    int RUNS = total/size;
    start = clock();
    for(int run=0; run<RUNS; run++) {
        /* vector triad */
        for(int ind=0; ind<size; ind++) {
            a[ind] = b[ind] + c[ind]*d[ind];
            if(((double)ind)==333.333)
                dummy(ind);
        }
    }
    end = clock();
    wallTime = (end - start)/((double)CLOCKS_PER_SEC);
    /* Avg throughput */
    double throughput = ((double)sizeof(double) * total * 2)/
```

total = 2^{29}

for s = 2^3 to 2^{29}

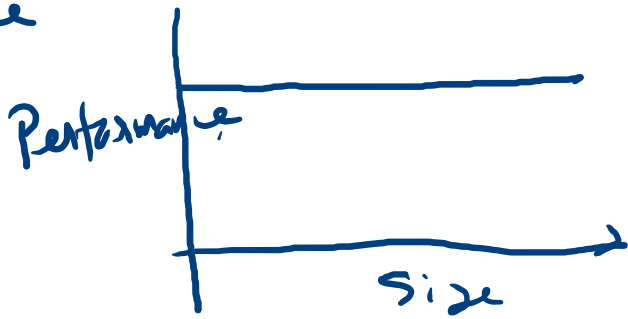
→ 2^3 to 2^4

→ 2^3 to 2^5

total = Runs * Size

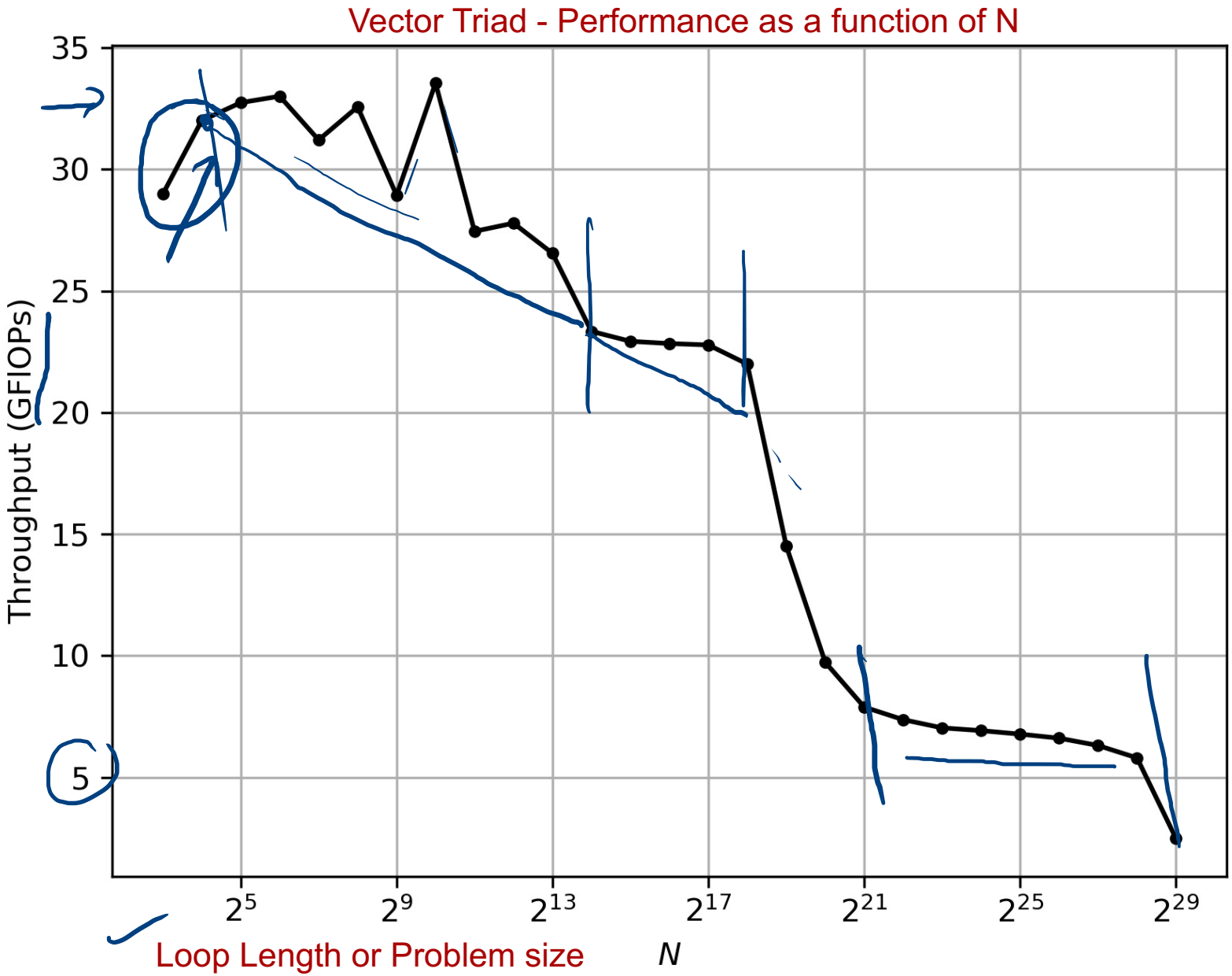
Size ↑ ↓ Runs

Something which is never true



Memory = (total * 4 * 8 bytes) / Walltime

3


Vector Triad - Performance as a function of N
Throughput (GFlOPs)
35
30
25
20
15
10
5
2^5
2^9
2^{13}
2^{17}
2^{21}
2^{25}
2^{29}
Loop Length or Problem size
N

Most important limitation – data access !!

Loop based code → large amount of data movement in and out of the CPU.
Concern : underutilization of on-chip resources.

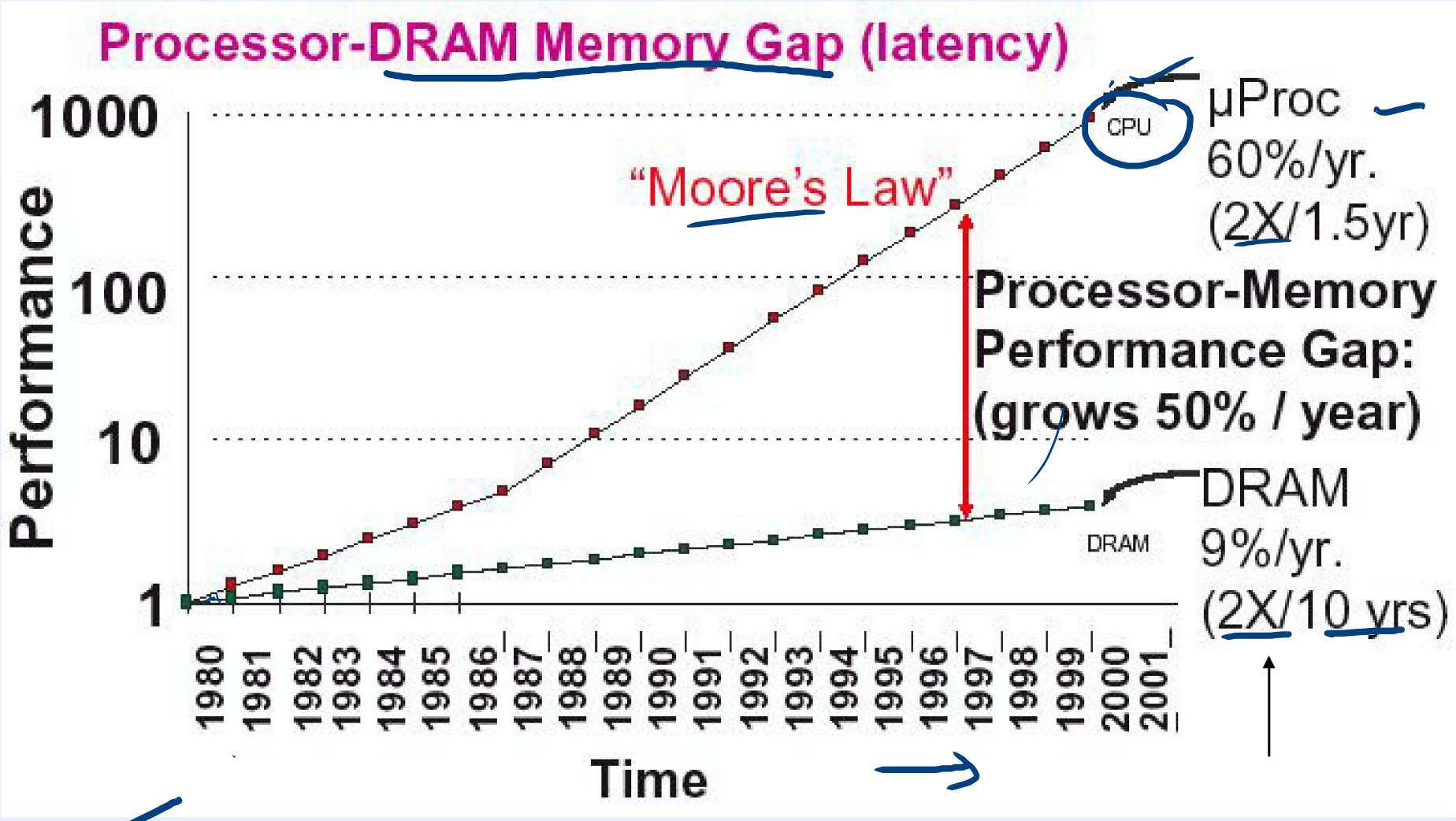
Several data paths → range of bandwidths and latencies.

Objective : reducing traffic over slow data paths.

Microbenchmarking

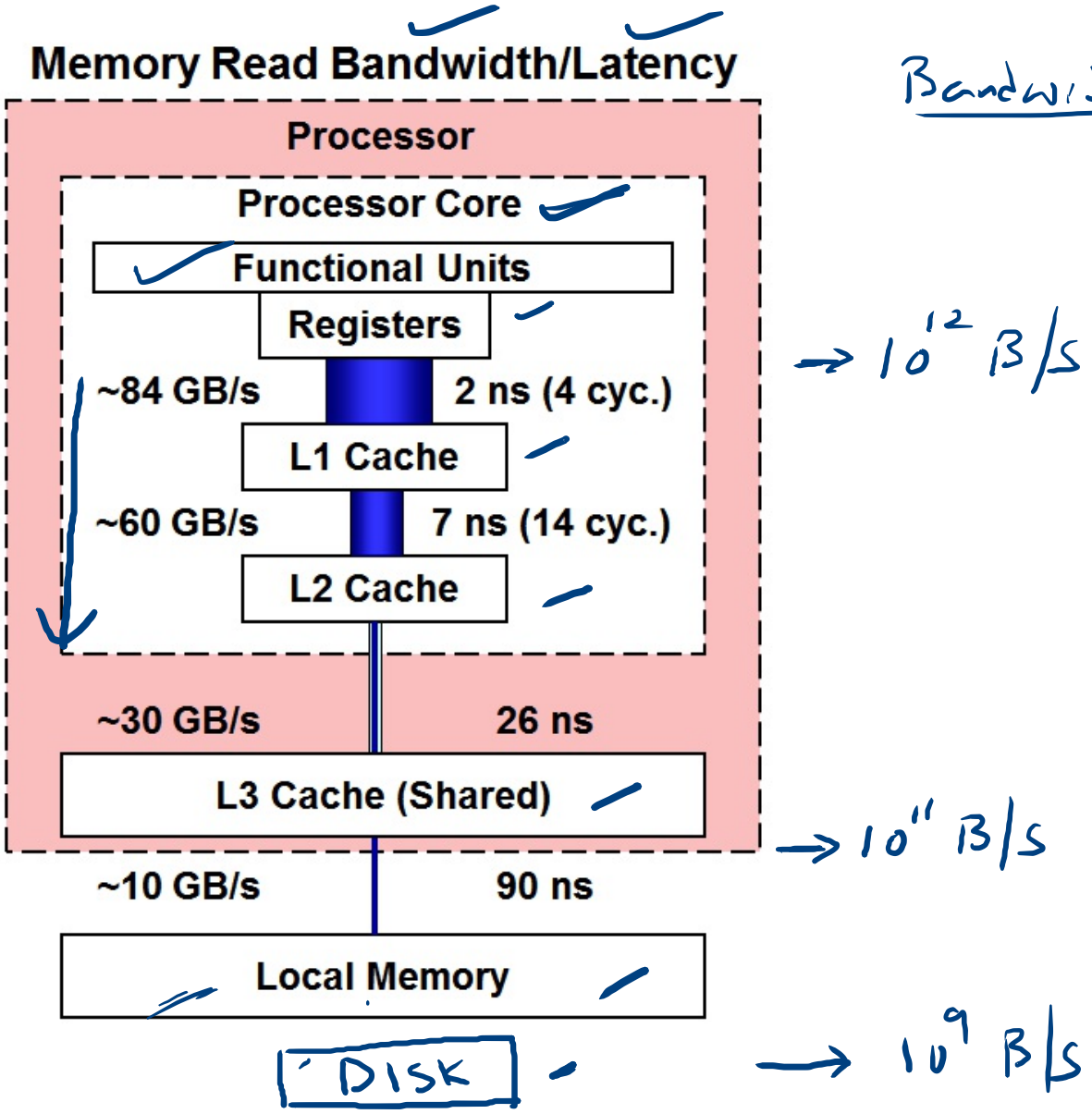
- **Probing of the memory hierarchy**
- **Saturation effects in cache and memory**
- **Typical overheads?**

What's Driving Parallel Computing Architecture?



von Neumann bottleneck!! (memory wall)

Latency (Sec)



Bandwidth.

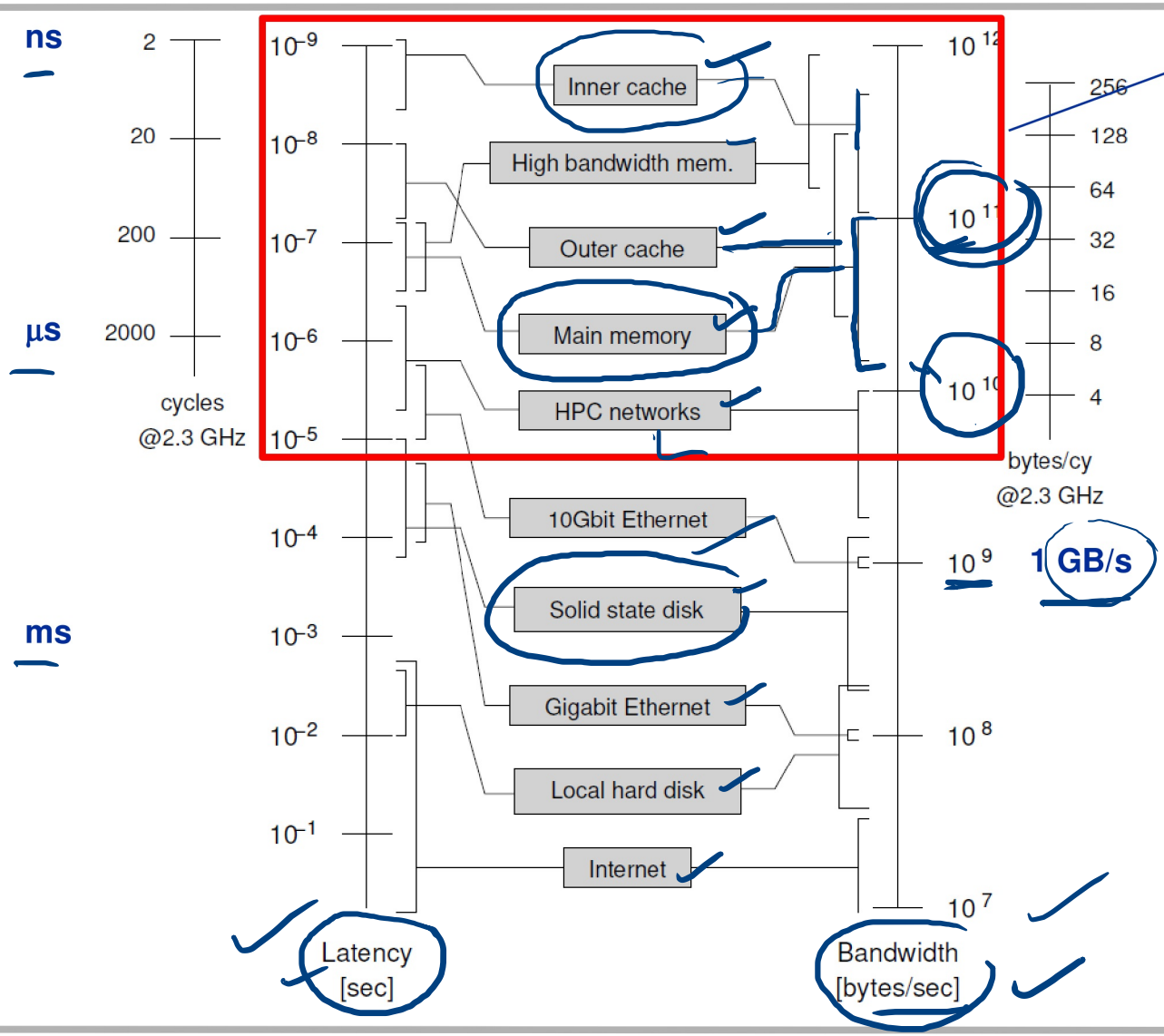
→ 10^{12} B/s

→ 10^{11} B/s

→ 10^{9} B/s

https://cvw.cac.cornell.edu/codeopt/memtimes

Latency and bandwidth in modern computer environments



HPC plays here

Avoiding slow data paths is the key to most performance optimizations!

But how "slow" are these data paths anyway?

Code balance

Code balance of a loop = data traffic/ floating point operations

Reciprocal of code balance → Computational Intensity.

Data → load, store and execution of operation.

Data traffic → performance limiting data path (hardware dependent).

Balance analysis

Theoretical performance of loop based codes.

Machine balance (processor chip) =
memory bandwidth / peak performance

Data Path	Bandwidth	Balance (units)
cache	10^{11}	?
RAM	10^{10}	?
interconnect		?
Disk	10^{9}	?

Machine balance will decrease or increase in future ?

Typical historical trend (balance) ??

Good Code ✓ B_c ↓ ↑ ✗

B_m is fixed

$\frac{B_m}{B_c}$ will increase for good log(x)

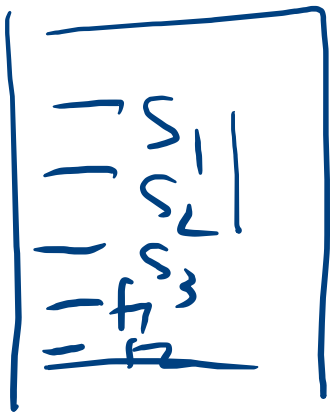
Optimizing a code → There is no alternative to knowing what is going on between your code and the hardware. → Performance Modeling.

Profiling

Profiling → information about program's behavior.
Runtime → "hot spots"
Hot spots → performance bottleneck → optimization.

Function and line based profiling.

Function profiling → (e.g. gprof from GNU)
→ Flat function profile and callgraph profile.



gprof

To get a break up of time taken by each subroutines/functions of the code :

compile with **f95 -pg file.f**

gcc - file.c

after running the executable (a.out) we will get a file **gmon.out**

We have to give the command **gprof --line a.out gmon.out** to get this breakup times.

→ “hotspots”

Terminology

Peak Performance. → for benchmarking.

Performance matrices →

Low level benchmarking → program to understand the chief performance characteristics of a processor/system.

How to do “time measurement” for different sections of the code. → “elapsed time” ?

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
48.85	19.47	19.47	3000	0.01	0.01	adv_efield_vel_
23.29	28.75	9.28	3000	0.00	0.00	rms_
12.15	33.59	4.84	3000	0.00	0.00	inc_efield_
11.87	38.32	4.73	3000	0.00	0.00	mr_mur_
3.77	39.82	1.50	3000	0.00	0.00	adv_hfield_
0.08	39.85	0.03	4	0.01	0.01	elec_dens_
0.03	39.86	0.01	394416	0.00	0.00	fioniz_
0.00	39.86	0.00	1	0.00	39.86	MAIN__
0.00	39.86	0.00	1	0.00	0.00	setup_

Old code - time taken (39.86 seconds)

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
52.76	14.04	14.04	3000	0.00	0.00	adv_efield_vel_
17.47	18.70	4.65	3000	0.00	0.00	rms_
17.40	23.33	4.63	3000	0.00	0.00	inc_efield_
6.61	25.09	1.76	3000	0.00	0.00	mr_mur_
5.52	26.56	1.47	3000	0.00	0.00	adv_hfield_
0.08	26.58	0.02	394416	0.00	0.00	fioniz_
0.04	26.59	0.01	4	0.00	0.01	elec_dens_
0.00	26.59	0.00	1	0.00	26.59	MAIN__
0.00	26.59	0.00	1	0.00	0.00	setup_

New code - time taken (26.59 seconds)

% time - the percentage of the total running time of the program used by this function.

Cumulative upto this function.

self - the number of seconds accounted for by this function alone.
seconds

calls - the number of times this function was invoked.

self - the average number of milliseconds spent in this function per call .
ms/call

total - the average number of ms spent in this function and its descendents per call
ms/call

name - the name of the function.

Callgraph profile / butterfly graph

Runtime profile of a function → several different callers.

```
index % time    self  children    called     name
                0.00   26.59       1/1          main [2]
[1]    100.0    0.00   26.59         1          MAIN__ [1]
               14.04    0.00    3000/3000        adv_efield_vel_ [3]
                4.65    0.00    3000/3000        rms_ [4]
                4.63    0.00    3000/3000        inc_efield_ [5]
                1.76    0.00    3000/3000        mr_mur_ [6]
                1.47    0.00    3000/3000        adv_hfield_ [7]
                0.01    0.02       4/4           elec_dens_ [8]
                0.00    0.00       1/1           setup_ [10]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   26.59                     main [2]
                0.00   26.59       1/1           MAIN__ [1]
-----------------------------------------------
               14.04    0.00    3000/3000        MAIN__ [1]
[3]     52.8   14.04    0.00    3000             adv_efield_vel_ [3]
```

Contd ….

```
-----------------------------------------------
                1.50    0.00    3000/3000        MAIN__ [1]
[7]      3.8    1.50    0.00    3000         adv_hfield_ [7]
-----------------------------------------------
                0.03    0.01       4/4           MAIN__ [1]
[8]      0.1    0.03    0.01       4          elec_dens_ [8]
                0.01    0.00  394416/394416      fioniz_ [9]
-----------------------------------------------
                0.01    0.00  394416/394416      elec_dens_ [8]
[9]      0.0    0.01    0.00  394416          fioniz_ [9]
-----------------------------------------------
                0.00    0.00       1/1           MAIN__ [1]
[10]     0.0    0.00    0.00       1          setup_ [10]
```