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

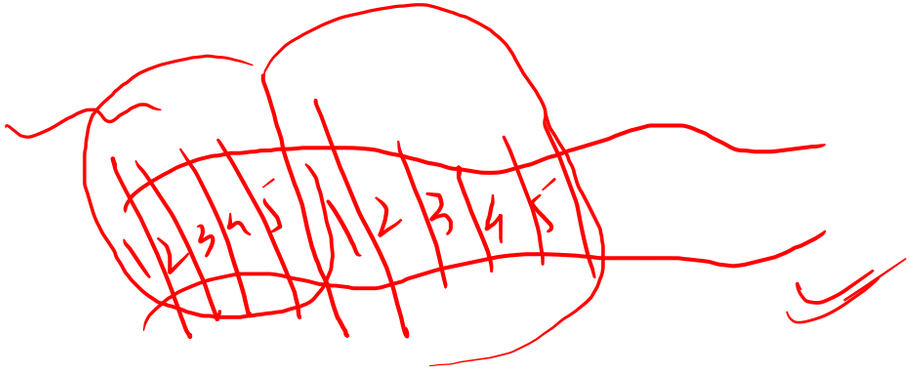
Resource → Searching tool

5 hrs

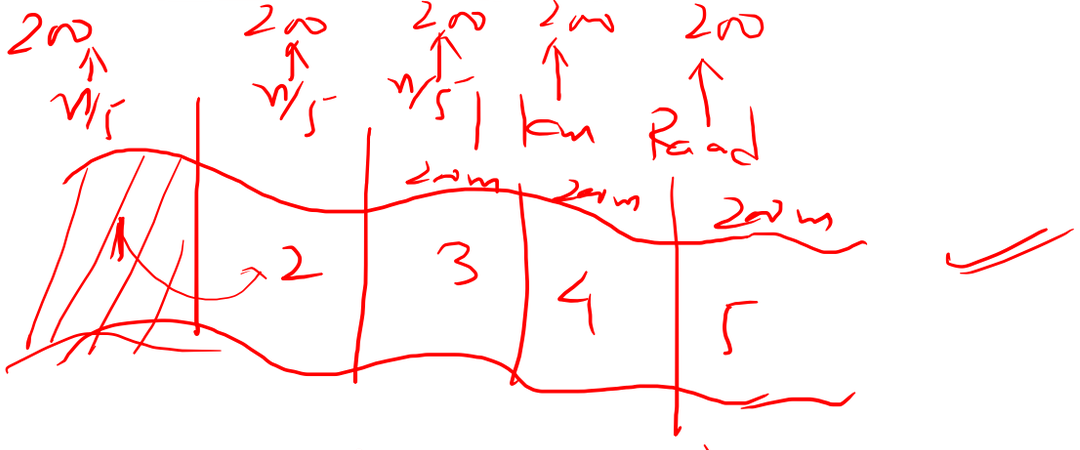
Adding n numbers

for i = 1 n

end

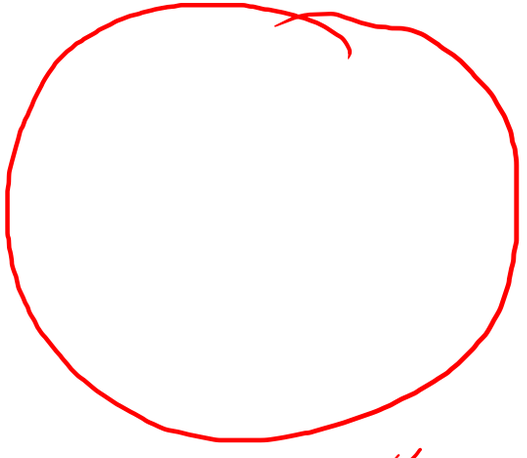


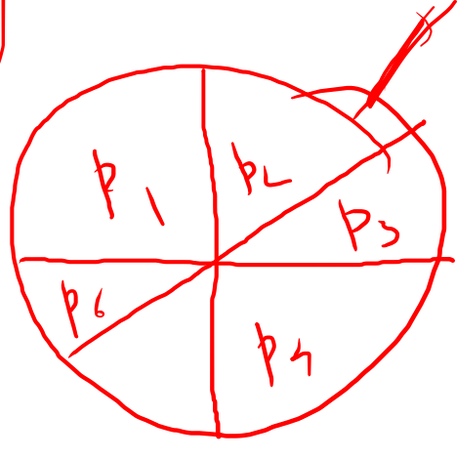
Thinking in Parallel



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

for i = 1, 200	for i = 201, 400
partial Avg	partial Avg





Dig a hole

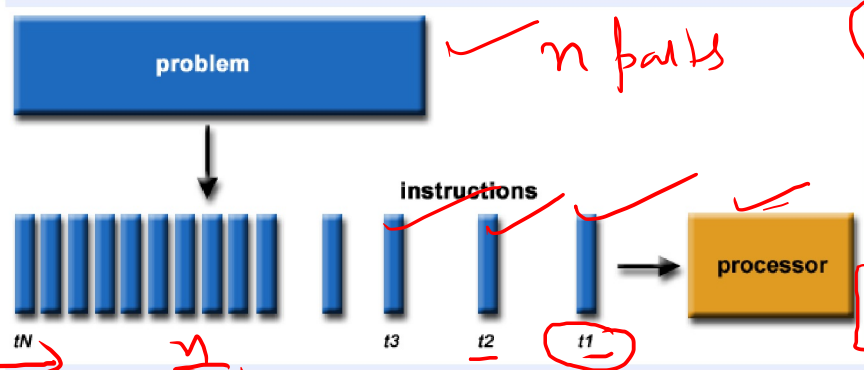
1m diameter
&
1m deep.

5 philosophers

Dependency

10 processes

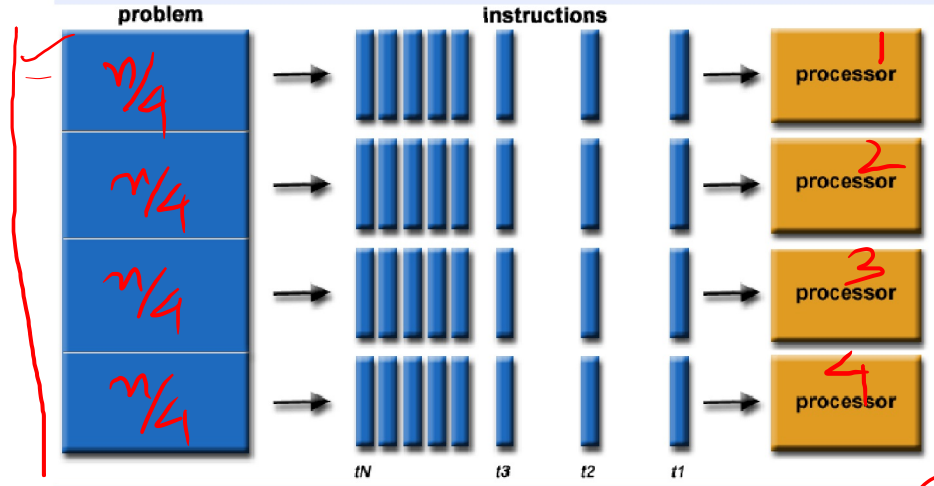
Serial vs Parallel



Serial

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

Improve Latency.



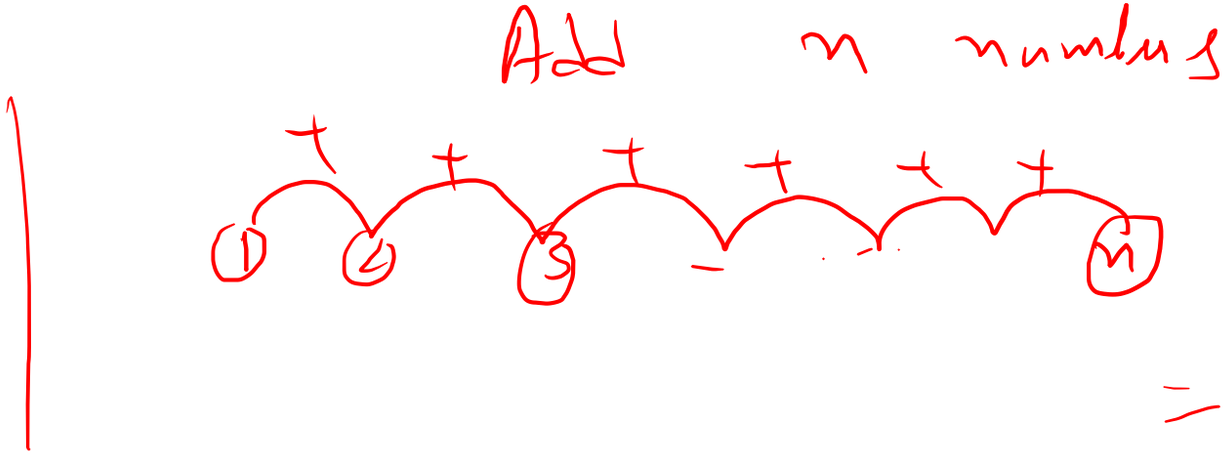
Parallel: multiple compute resources.

- 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.

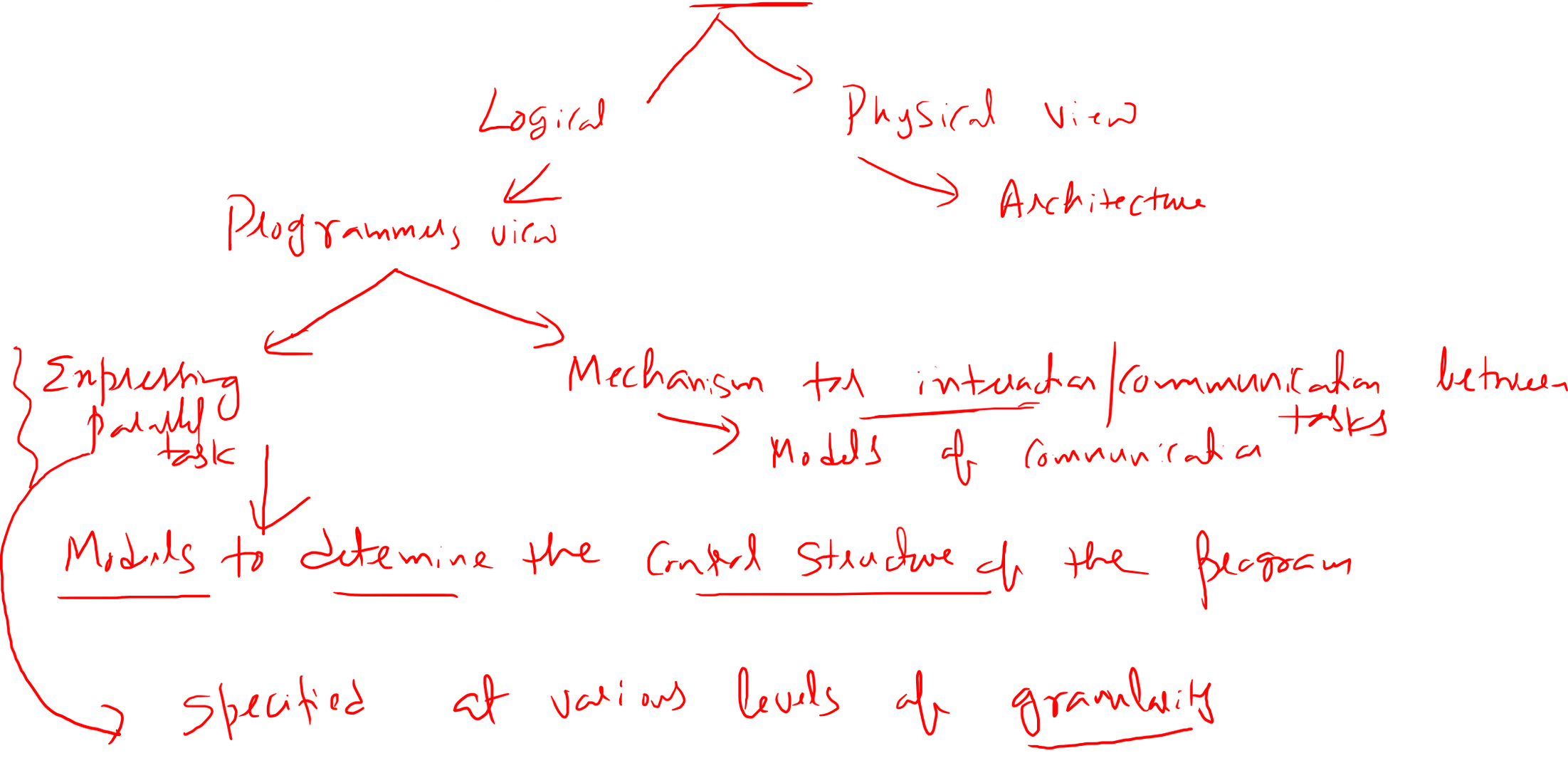
Improve throughput.

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



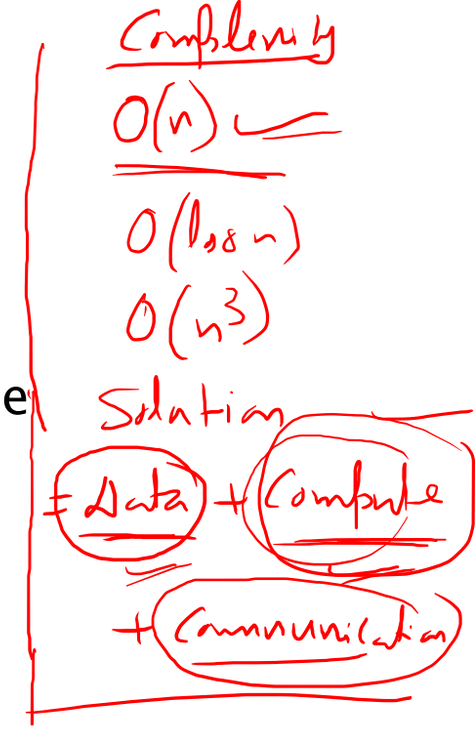
Organization of Parallel Platforms

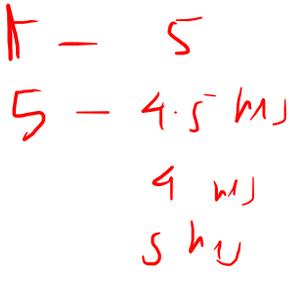


Theory of parallel algorithm design or theory of parallel computing is based on abstract concepts of time and memory which may ignore real life constraints for simplicity, and therefore not take into account non-deterministic and hardware factors

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.





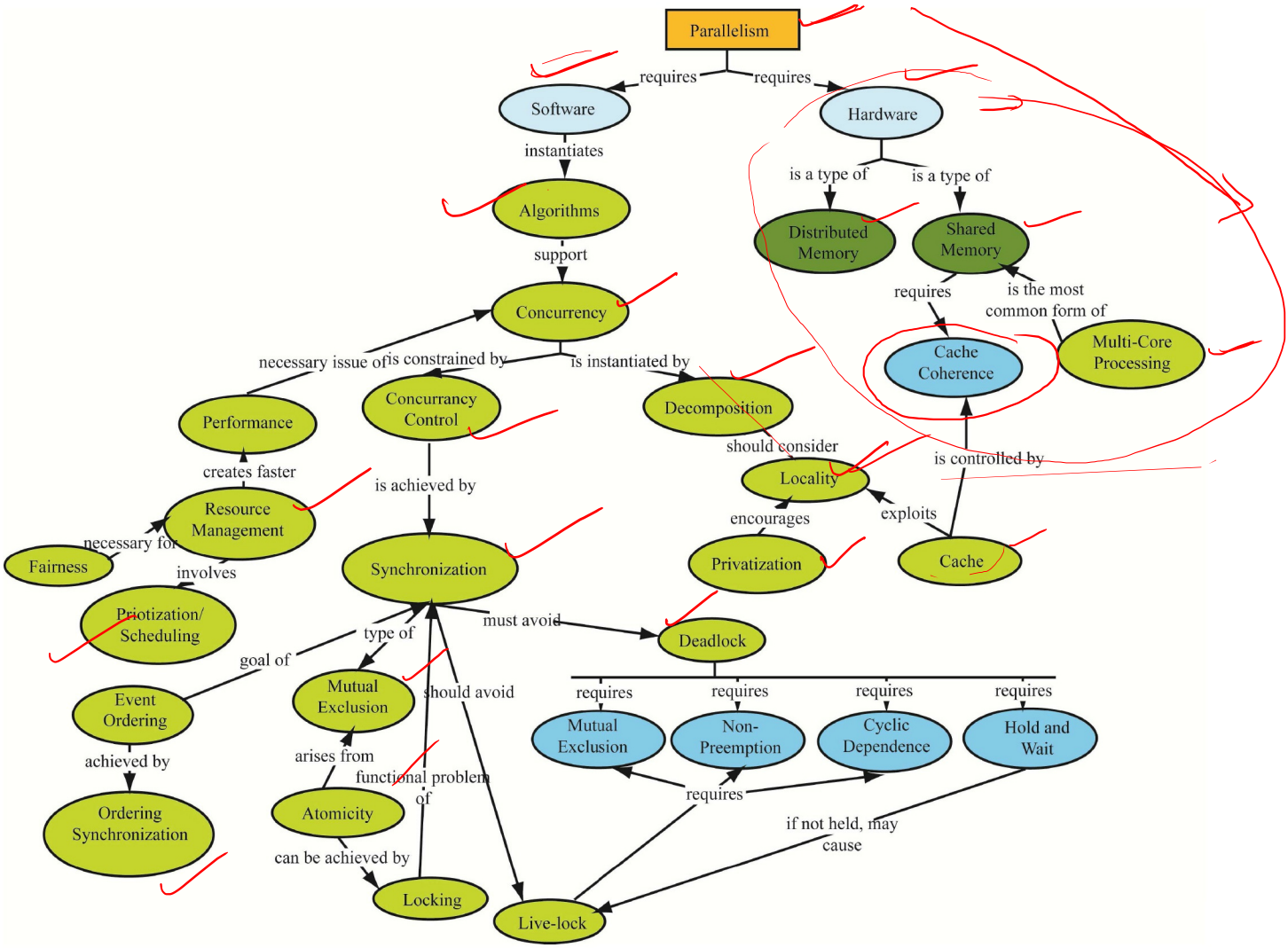


Figure 1. Concept map for parallel computing. Adapted from [9].

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.

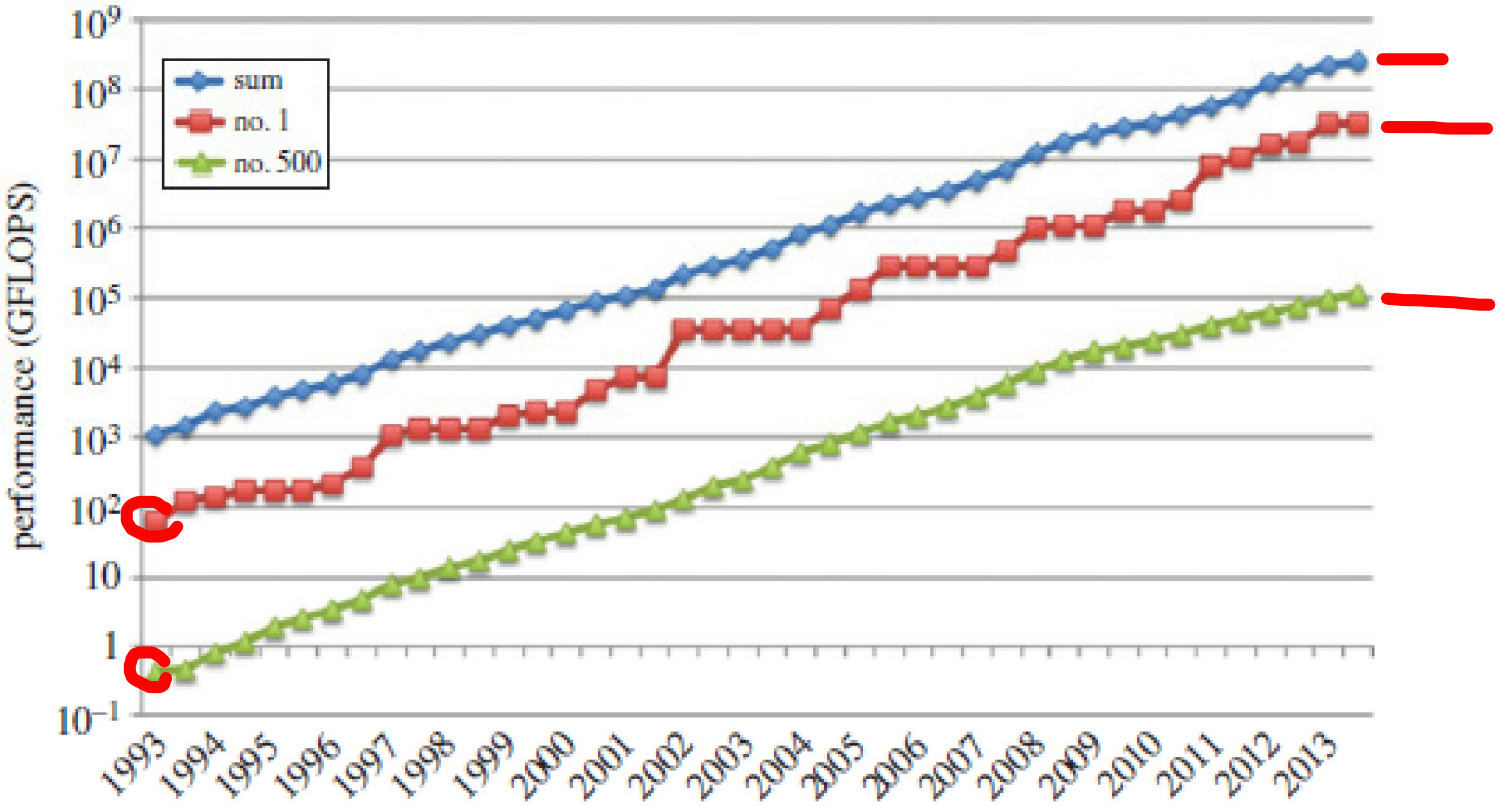


Figure 1. Data showing performance of Top500 supercomputers on the LINPACK test over the past 20 years. The top line is the sum of the top 500 systems, the middle line is no. 1, and the bottom line is no. 500 [1]. (Online version in colour.)

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

HPC trends

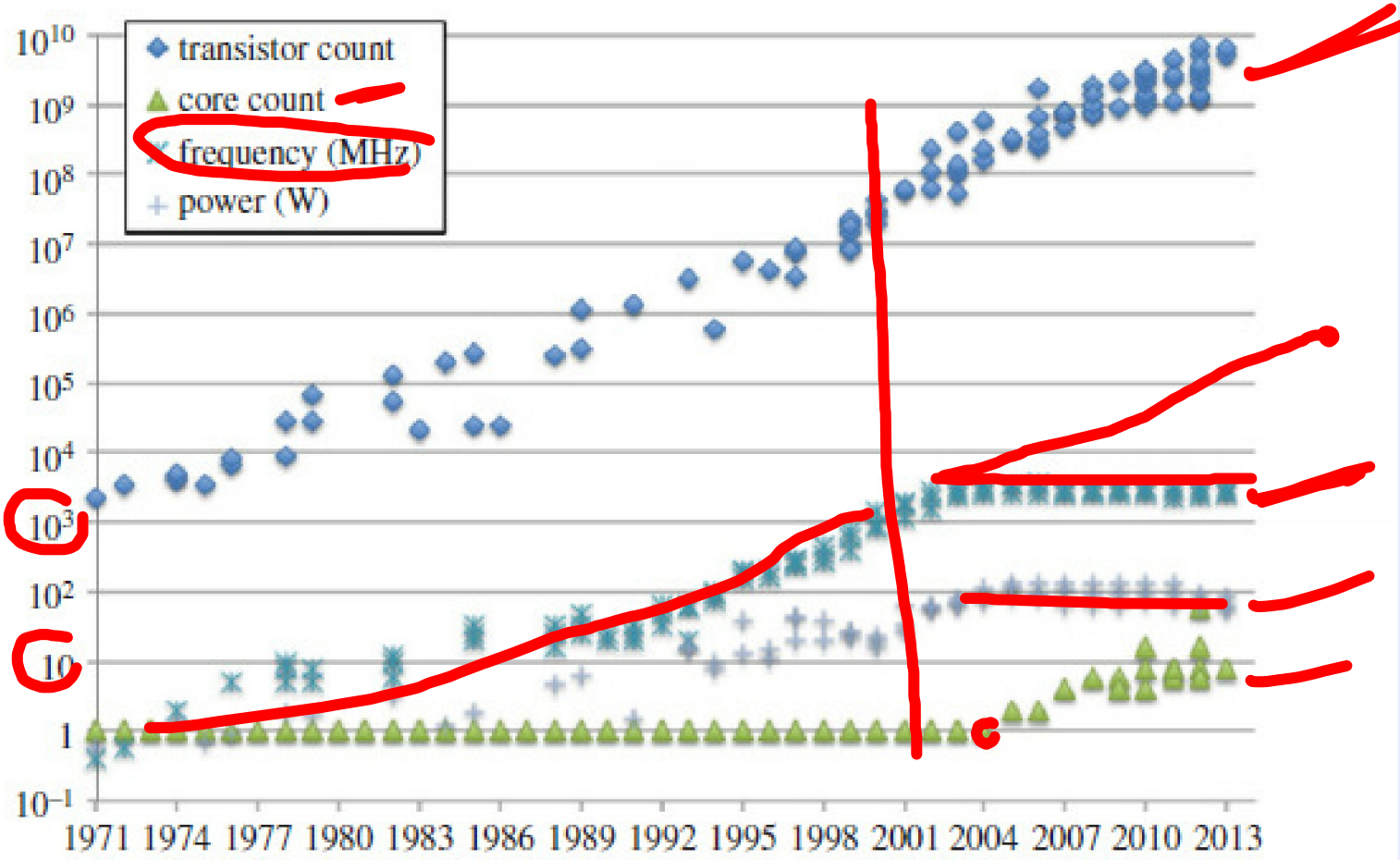


Figure 3. Evolution in transistor count, clock frequency, number of cores and power consumption for processors over the past 40 years. (Online version in colour.)

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

HPC Applications

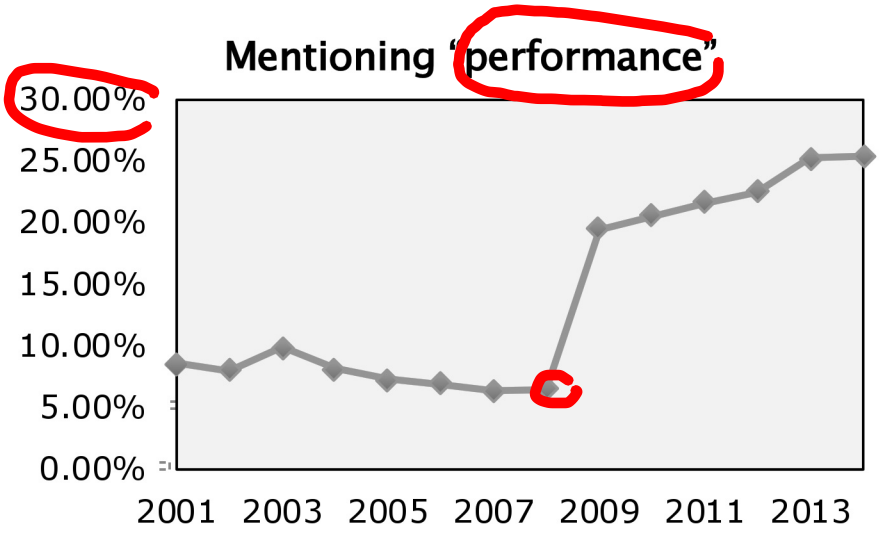
Science and Engineering:

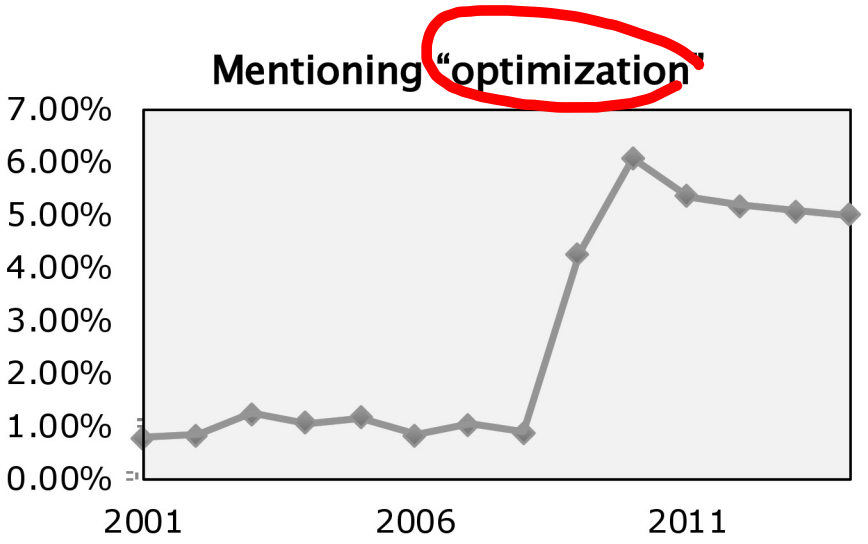
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion.
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Mechanical Engineering
- Electrical Engineering, Microelectronics
- Computer Science, Mathematics
- Defense, Weapons

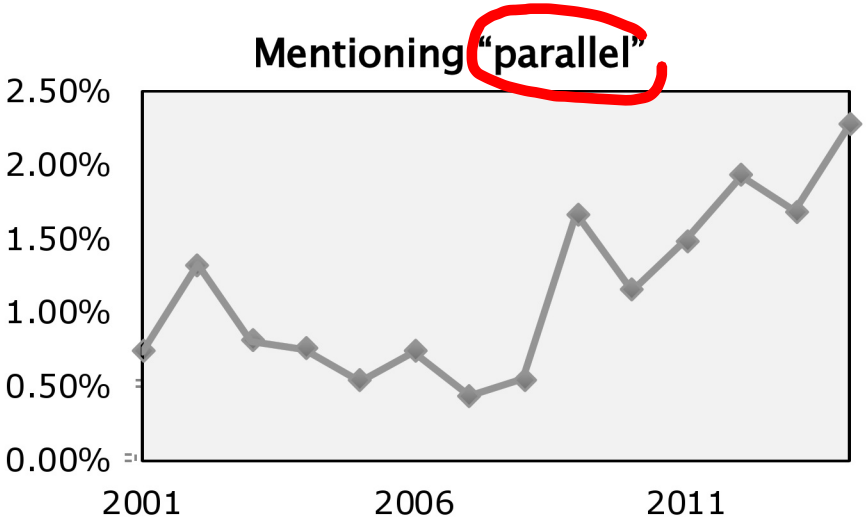
Industrial and Commercial:

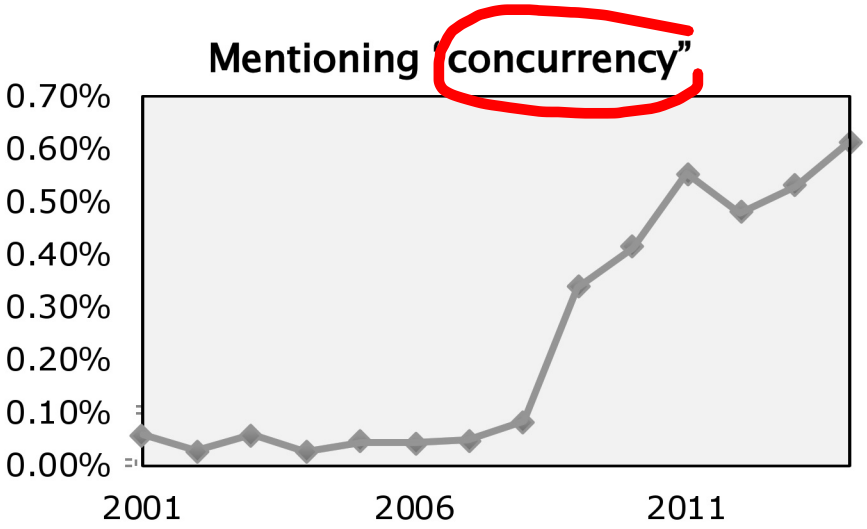
- Databases, data mining
- Oil exploration
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial modeling
- Entertainment industry
- Web search engines, web based business services

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 !

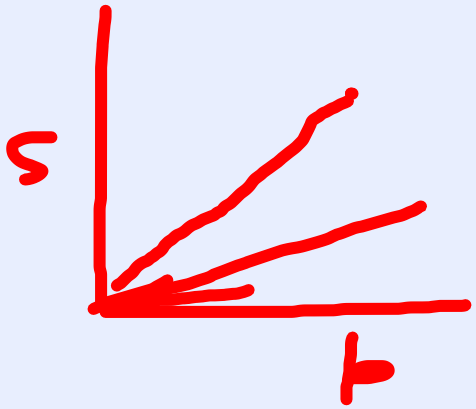
High Time to learn Parallel Programming →

- 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.

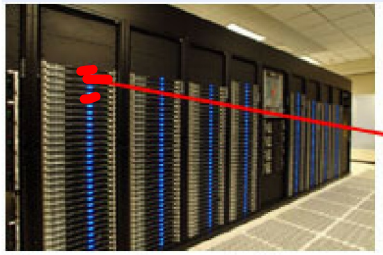
Supercomputer = several nodes → HPC

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)?

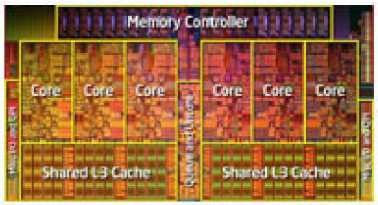
Terminology --- visual

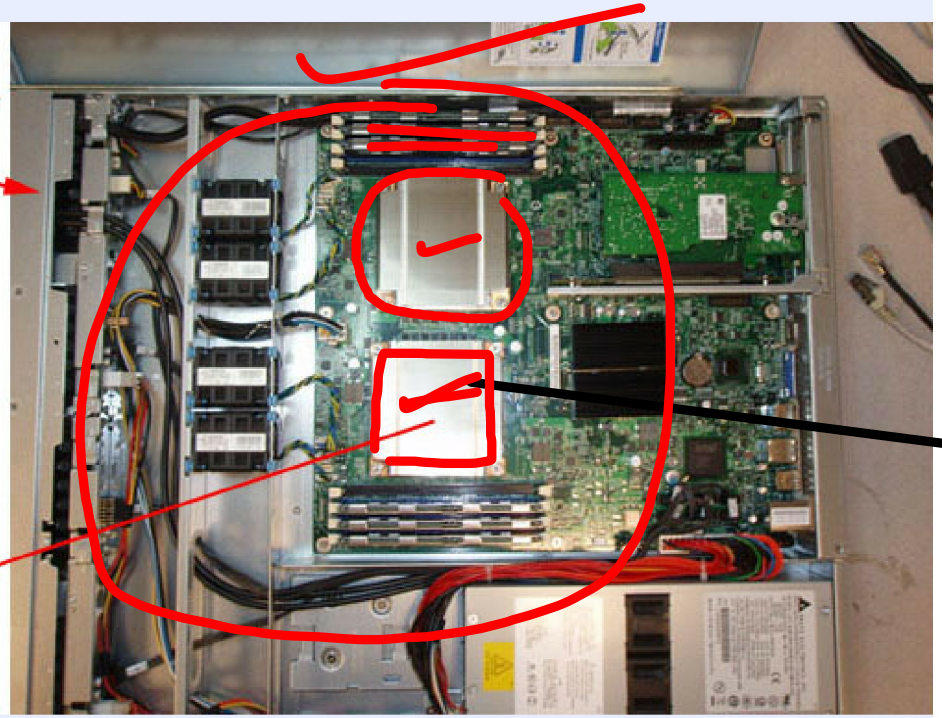


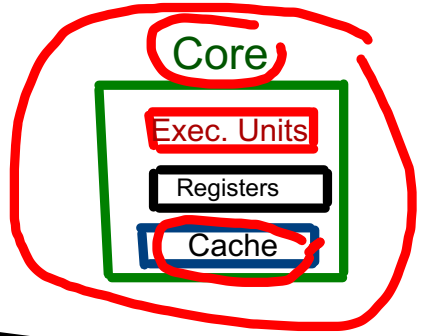
Supercomputer - each blue light is a node

Single O/S — Node - standalone Von Neumann computer

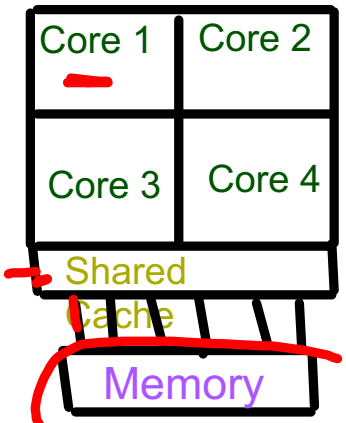
CPU / Processor / Socket - each has multiple cores / processors.







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

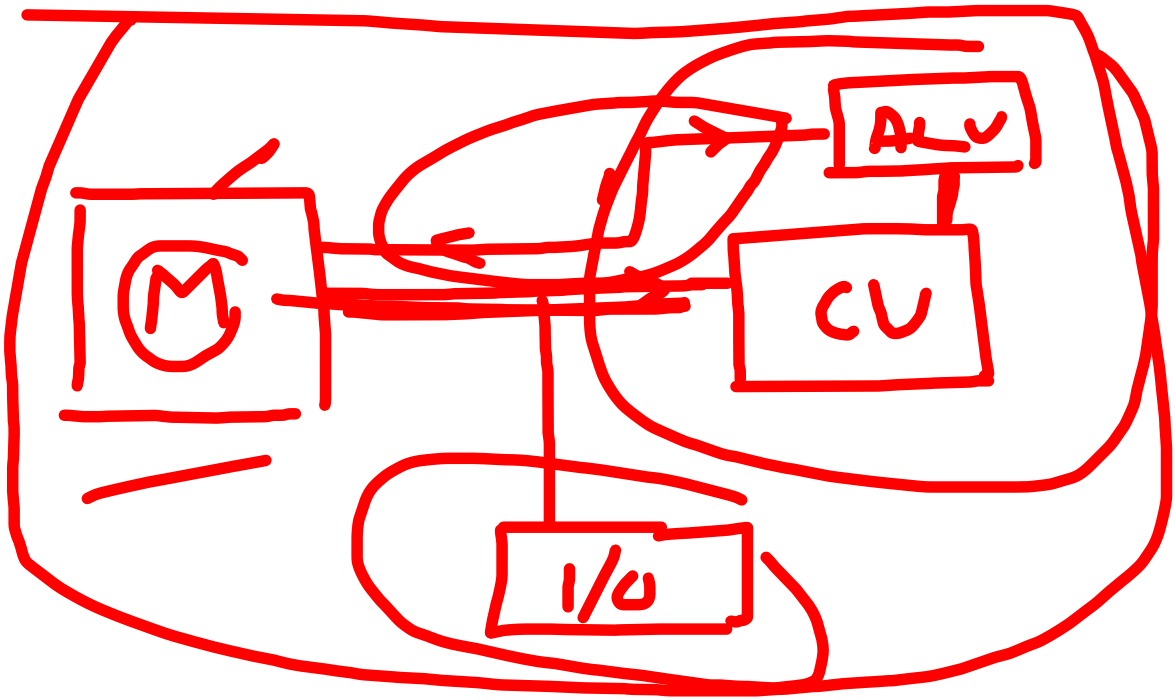
You ***must*** re-think your algorithms to be parallel !

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)
```

$N = 10^6$

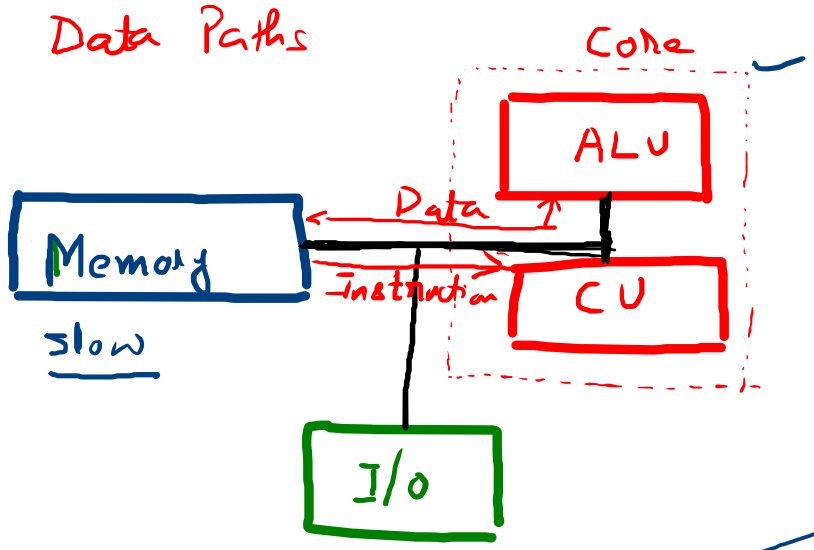
$Perf = \frac{10^6}{Time(s)}$

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.
L> 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}

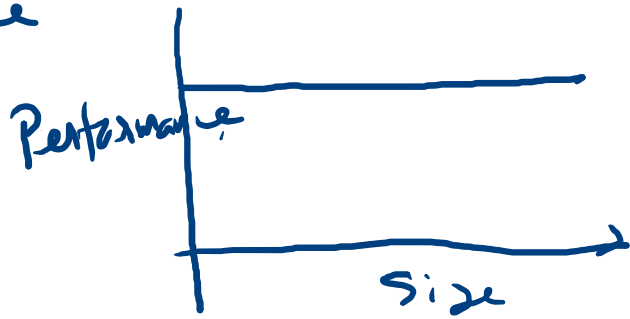
→ 2^3 to 2^4

→ 2^3 to 2^5

total = Runs * Size

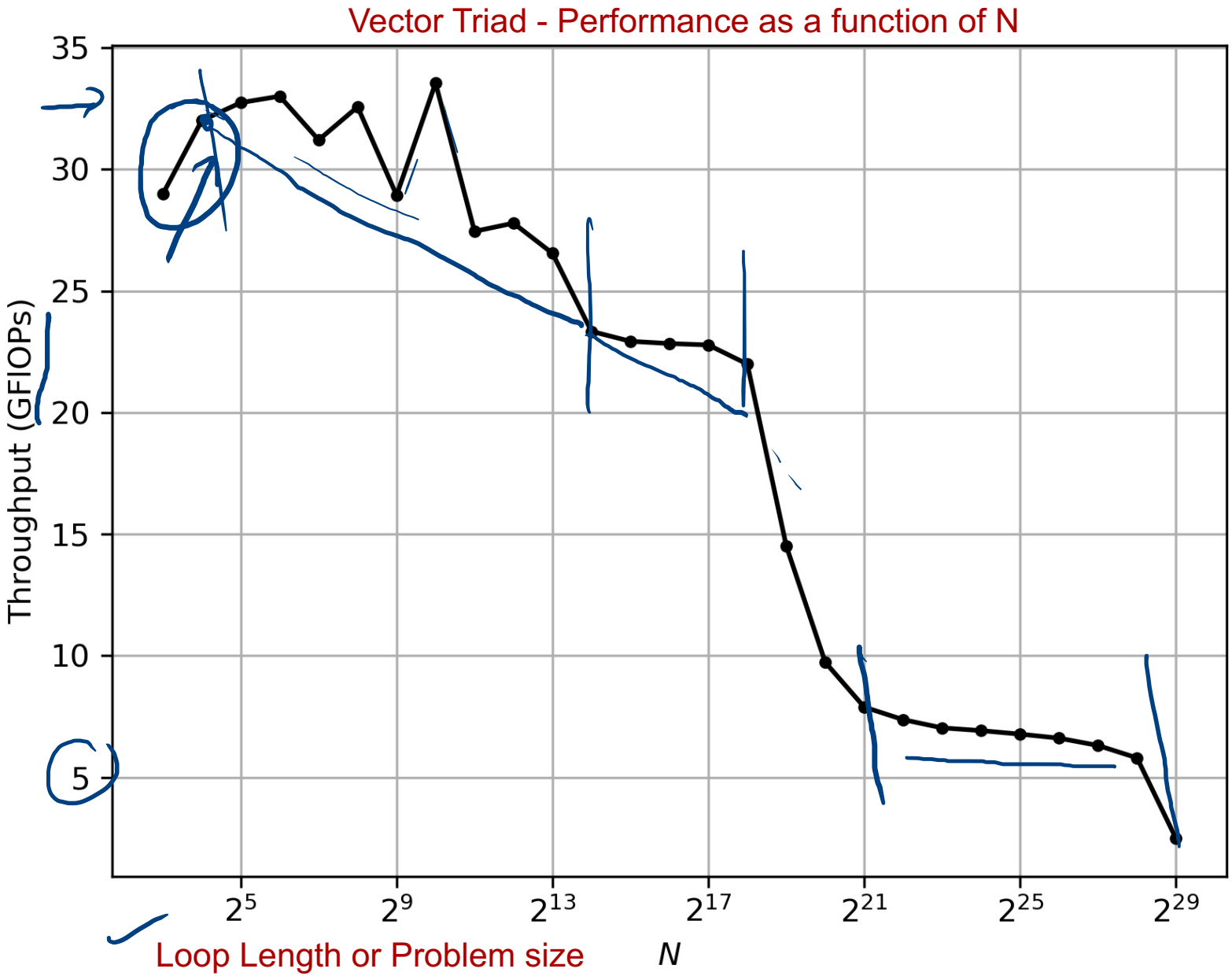
Size ↑ ↓ Runs

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.

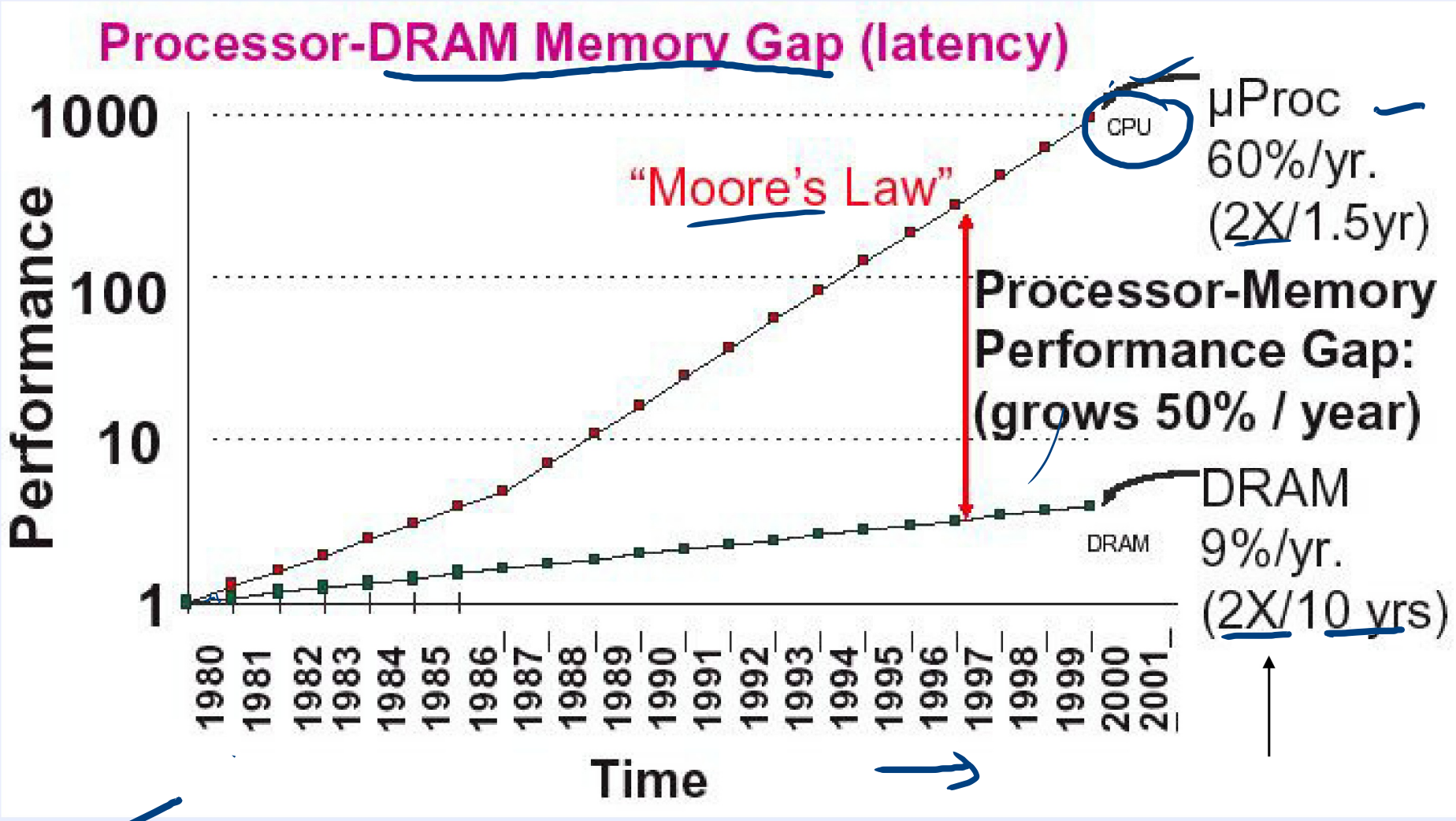
Several data paths → range of bandwidths and latencies.

Objective : reducing traffic over slow data paths.

Microbenchmarking

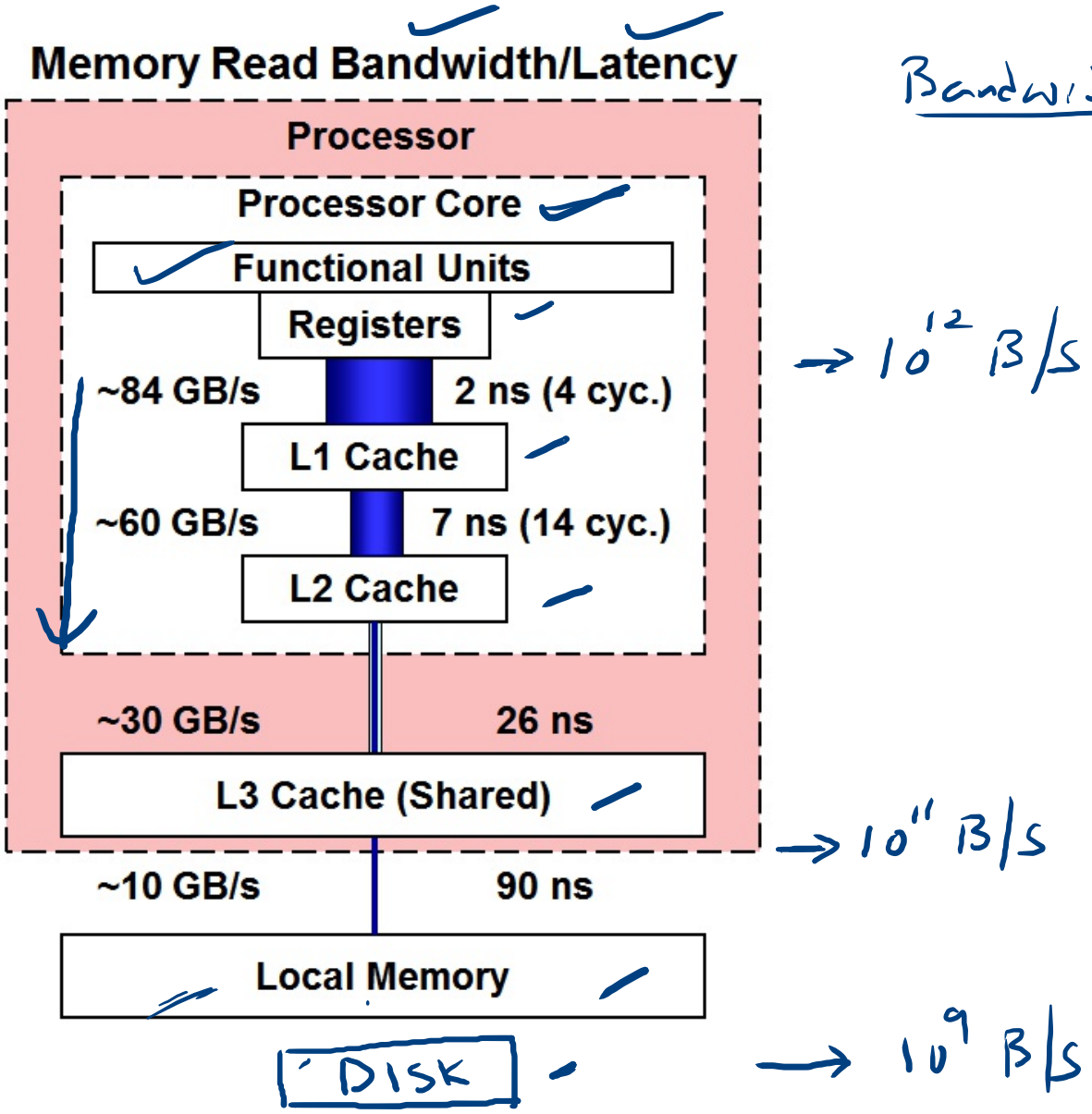
- **Probing of the memory hierarchy**
- **Saturation effects in cache and memory**
- **Typical overheads?**

What's Driving Parallel Computing Architecture?



von Neumann bottleneck!! (memory wall)

Latency (Sec)



Bandwidth.

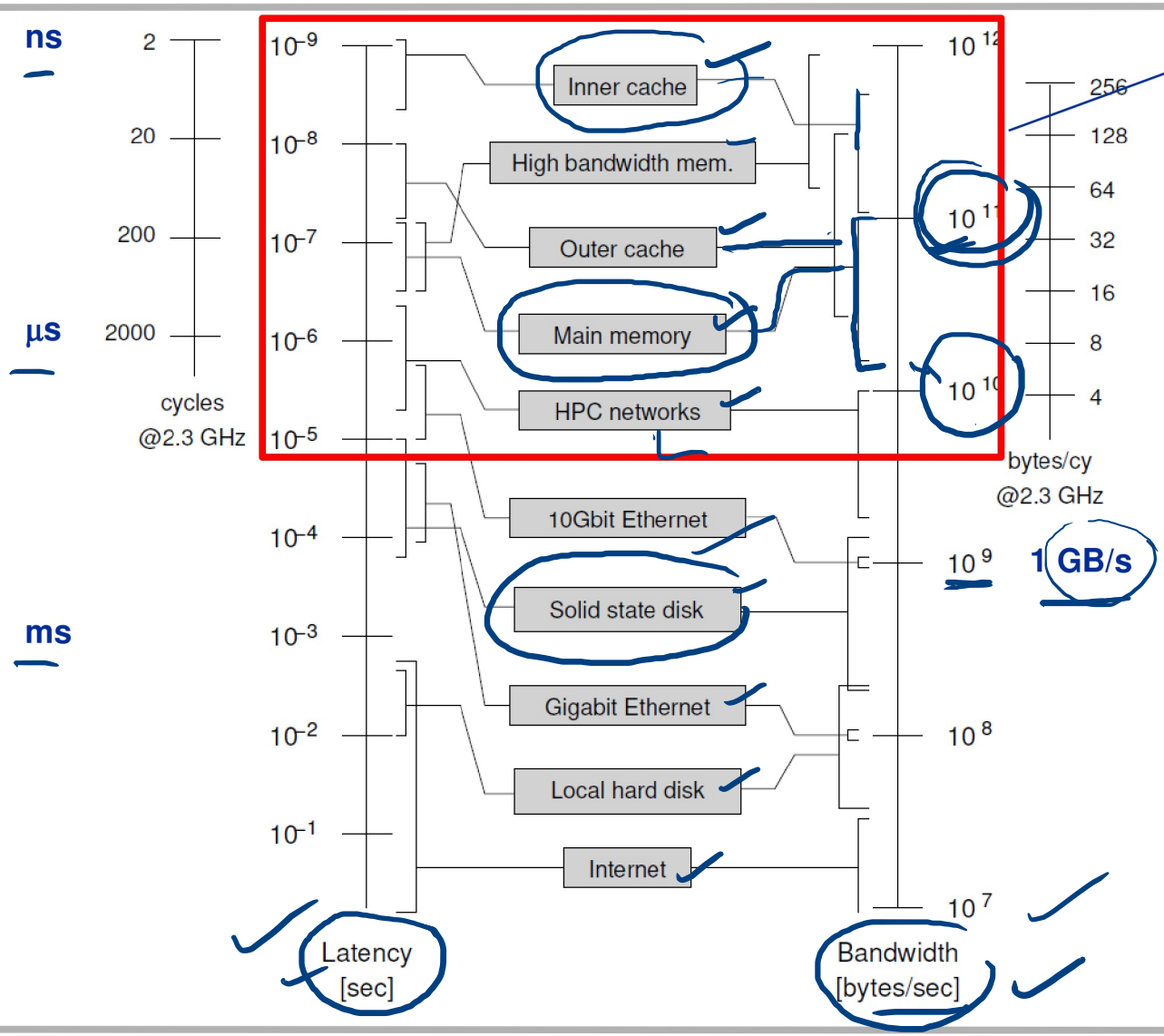
→ 10^{12} B/s

→ 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)

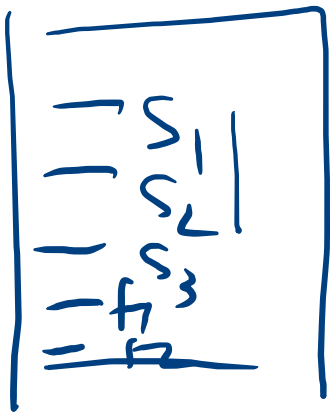
Optimizing a code → There is no alternative to knowing what is going on between your code and the hardware. → Performance Modeling.

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]
```

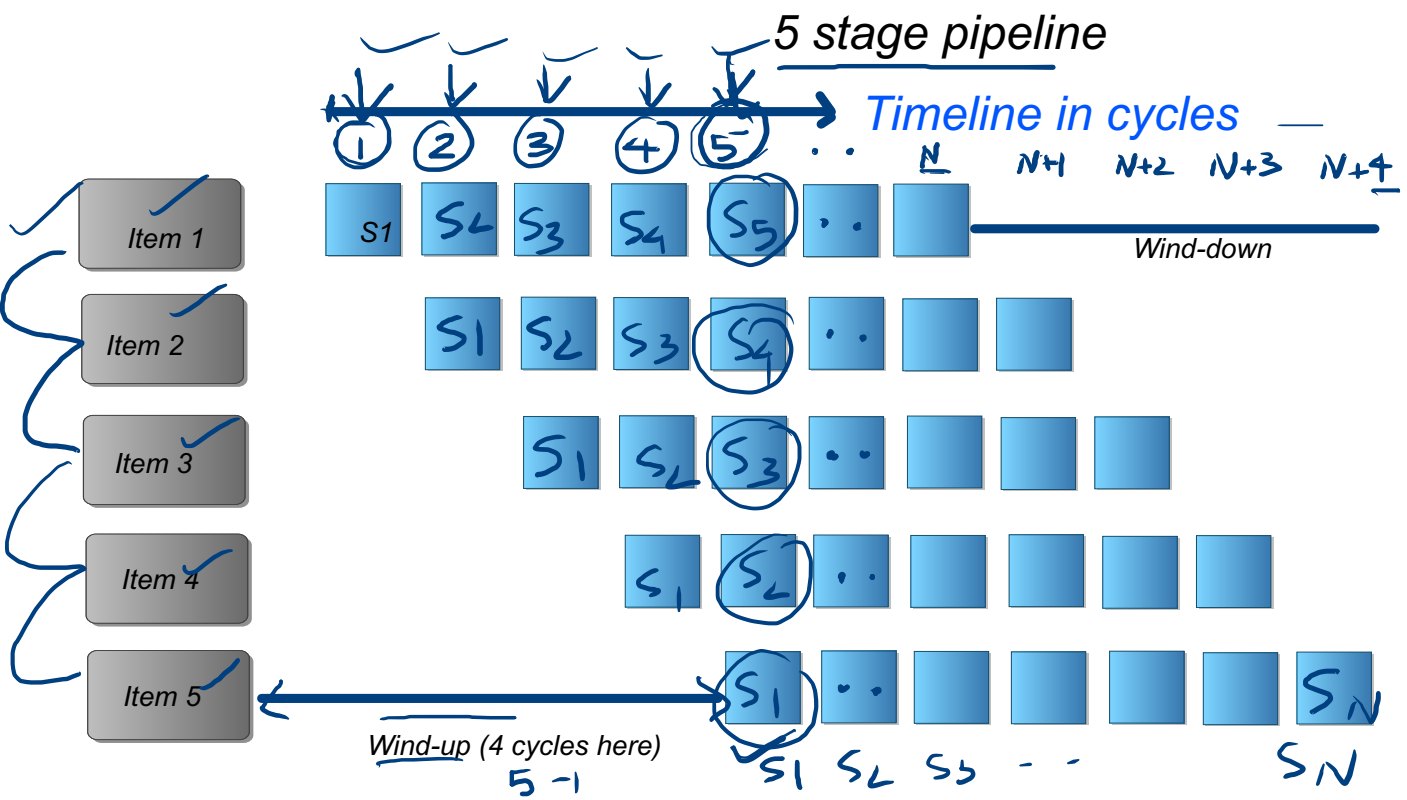
Contd ….

```
-----------------------------------------------
                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]
```

Lecture 4
CS301- High Performance Computing
Course Instructor : B. Chaudhury

To understanbd the results of vector triad , we need certain concepts i.e. techniques to improve application performance

* *Pipelined functional units*
* *superscalar architecture*
* *Data Parallelism through SIMD*
* *Out of order execution*
* *Larger Caches*



This pipeline has a latency (depth) of five cycles because after this 1st result is available

Food - some operation like Addition or Multiplication
Items/ dishes - sub-tasks to complete operation
Steps - stages of pipeline
N students - N operations (A[i]=B[i]+C[i] where i=1,N)
Simplest pipeline --> fetch (item 1) - decode (item 2) - execute (item 3)

* *Pipeline context- Students taking food in a queue (line)*
* *Total N students*
* *Complete Food (lunch) contains 5 items (dishes) - to be taken one after the other*
* *All students taking same food*
* *serving/taking each item is a task (step) - toal 5 steps required*
* *Workers (serving food) are highly skilled and specialized for a single task*
* *Each worker executes same work but works on different objects (serving same food item to different students)*
* *All tasks (steps) take around same time (1 cycle)*
* *Forwarding the partially-finished work to the next*

Pipelining of Functional units

Core can work on several independent instructions simultaneously (parallely) - working on 5 additions at the same time after the wind-up phase
One instruction gets finished at each cycle after the pipeline is full
Widely used in modern processors

Pipeline subtask - load, store, address calculation, decode, execute etc.

Job of compiler - arrange instructions in such a way as to make efficient use of all the different pipelines

Generalize : Pipeline of depth "m"

Total independent operations to be executed - N

for i = 1, N

Pipeline → Total time taken to finish (T_pipeline) in cycles - $m + N - 1$

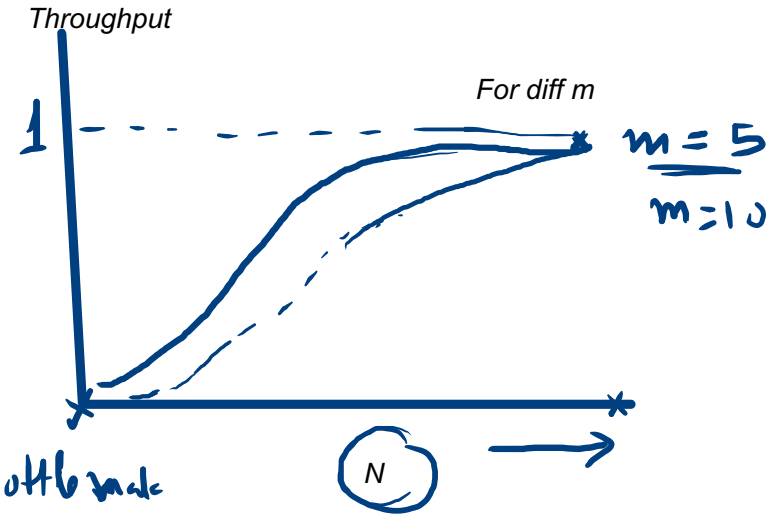
Unit without pipeline takes "m" cycles to generate a single result; how much for N results - mN

Speedup = $\frac{mN}{m+N-1} \approx m$ $\quad N \gg m$

Conclusion: Depth of pipeline decides speedup

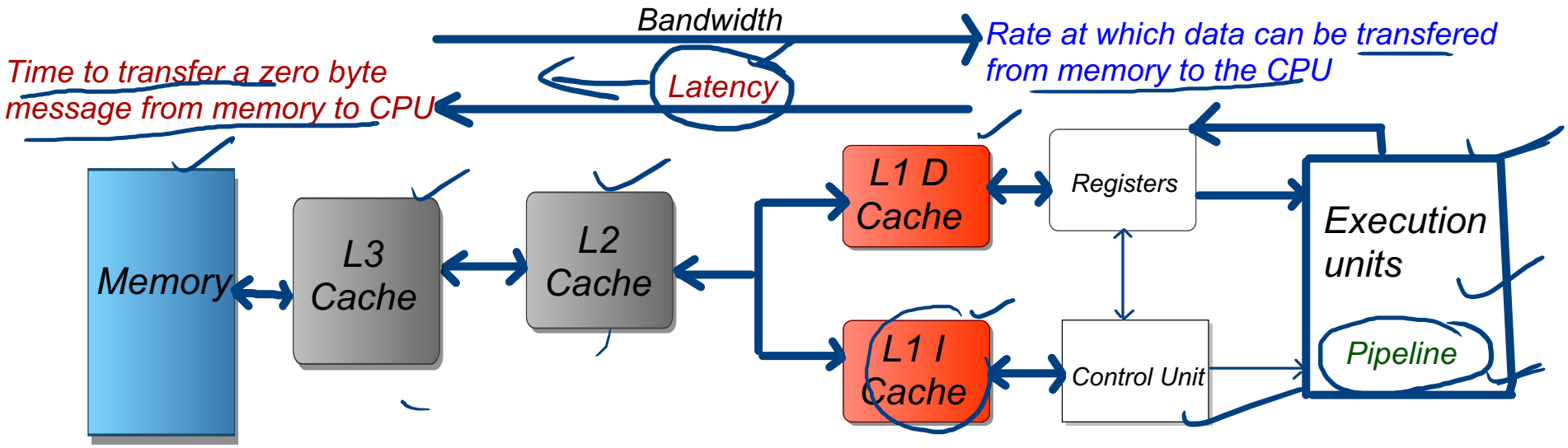
Throughput of the pipeline = $\frac{N}{N+m-1} = \frac{1}{1 + \frac{m-1}{N}}$

for large N $\therefore$ Throughput $\to 1$



& Conclusion: Tight loop (N is small) → pipeline bottleneck

Memory Hierarchies



DRAM Gap - increasing distance between CPU and memory in terms of latency and bandwidth
Cache alleviate the effects of DRAM gap.

CPU issues a read request to transfer data item to register, first level cache logic checks whether this item already exists in cache or not

* If it exists - cache hit (low latency)
* If it does not exist - cache miss (high latency) --> fetch from outer cache --> worse from memory

Instruction cache miss are rare events compared to data cache miss

* Either build a small and fast memory or a large and slow memory

* Multiplicity- multiple levels - the larger the slower. Typical caches - KBytes to MBytes

* Data transfers occur in a block (bursts) of single cache lines (~64 bytes) from one level to another level.

Goal-avoid slow data paths- optimization

Data transfer basic model in terms of latency and bandwidth - for different data paths
Data - N bytes
Latency - T_L (wait time)
Bandwidth - B bytes/sec (after data starts flowing)

Transfer time for message $T = T_L + N/B$

Effective bandwidth of data transfer $= N/T = N/(T_L + N/B)$

Application (algorithms)

** Access pattern - shows some locality of reference*

Data items loaded into cache gets used again before getting evicted --> Temporal locality (reuse in time)

Estimation of performance gain from cache:

Suppose cache is k times faster than memory - k comes from b oth latency and bandwidth

→ A[i] ↓
→ A[i]
→ A[i]

Cache reuse ratio (fraction of load or store that can be satisfied from cache because there was a recent load or store to the same address) - r

Access time to main memory =>Tm

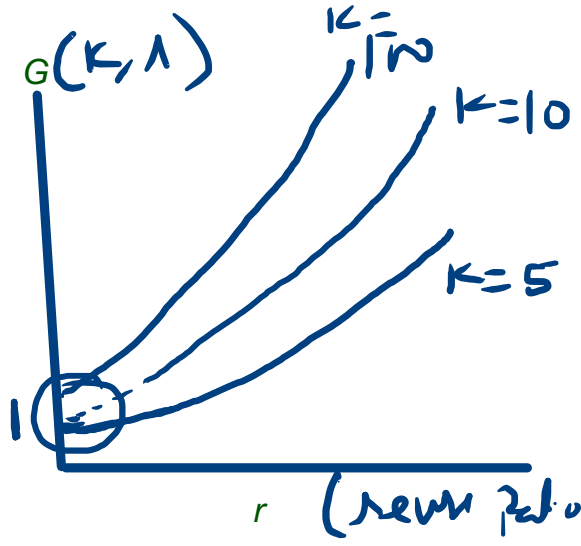
Cache access time Tc= T_m / K

Average access time (for some r) T_av = $rT_c + (1-r)T_m$

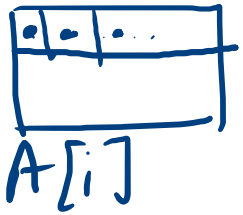
Performance Gain = $\frac{T_m}{rT_c+(1-r)T_m} = \frac{KT_c}{rT_c+(1-r)KT_c}$

$= \frac{K}{r+(1-r)K} = G(K, r)$

Reuse Ratio → 1 , Performance Gain = K



Limitation - many applications reuse ratio (temporal locality) is very low -->0 (for example image processing)

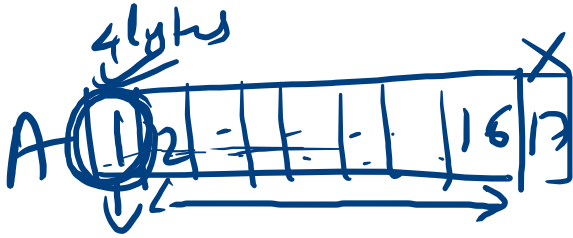


Each new load is expensive -- huge latency

To solve this issue --> concept of cache line comes into piucture

Goal - increase cache hit ratio (not cache reuse)

The cache line is fetched as a whole from memory, therfore latency penalty of a cache miss occurs only on the first miss.
Neighbouring items can be loaded from cache with much lower latency.



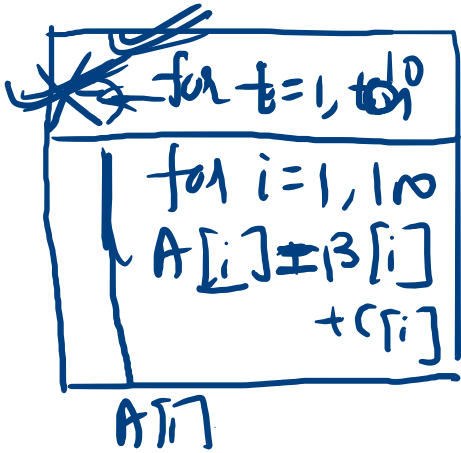
Whenever there is spatial locality --> latency problem can be reduced
Disadvatage - erratic data access pattern -- polluting the message bus

Suppose cache line length = 16

Spatial locality will fix the hit ratio at $= \frac{15}{16} = .94$

Algorithm - Performance is goverened by memory bandwith and latency - memory bound application

Stencil



Cost of cache miss

Profiles

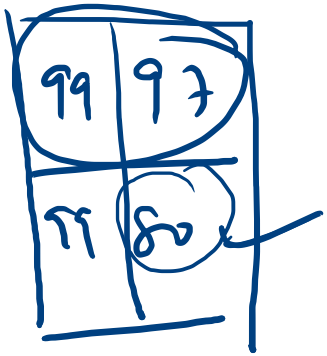
understand the difference between hit and miss
Hit time - time to deliver a line in the cache to processor (time to determine whether line is in cache included) - typical hit times - 1-2 clock cycles for L1 cache

Miss penalty - typically 10-100 of cycles

99% hit vs 97% hit --> do we need to worry and do something (increase hit)

99% vs 80%

Cache hit time - 1 cyle
Cache miss penalty - 100 cycles (L1 cache vs main memory)



Average Access time

97% hit → $\frac{97}{100} \times 1 + \frac{3}{100} \times 100 \sim 4$ cycles

99% hit → $\frac{99}{100} \times 1 + \frac{1}{100} \times 100 \sim 2$ cycles

80% hit → $\frac{80}{100} \times 1 + \frac{20}{100} \times 100 \sim 21$ cycles

2 times

2% diff → 2 times

→ 10 times

L1 → 1B Access latency → 4 cycles

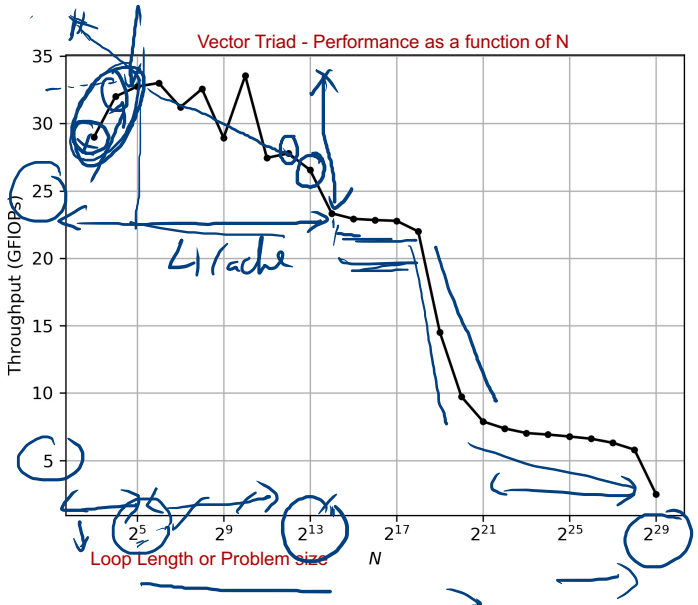
L2 → 1B

L3 → MB " → 30-40 cycles

Memory → GB " → few 100 cycles

Block Size - 64 bytes

Vector Triad interpretation




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

  int RUNS = total/size;

  for(int run=0; run<RUNS; run++) {

    for(int ind=0; ind<size; ind++) {
      a[ind] = b[ind] + c[ind]*d[ind];
```

Computation → 2 FLOPS

Data Accs → 4 data items

$2^3 \rightarrow 8 \times 32 = 256$ bytes

$2^5 \rightarrow 32$

Validity

$4 \times 8 = 32$

bytes

$2^{10} \rightarrow 1K$

$2^{20} - 1M$

$2^{30} - 1G$

$2^{13} \rightarrow$

$2^{10} \rightarrow 1K$

$2^{13} \rightarrow 8K$

4 values / 6 bytes

L1 128 KB → 256 KB

L2 64 KB

Size $- 2^3, 2^4, 2^5, 2^6$

$2^3 \rightarrow 8 \times 32 \rightarrow 256$ bytes

At small N processor pipeline is too long to be efficient

Vector triad performance analysis:

1. At small N, processor pipeline is too long to be efficient. (tight loop)

2. Reasonable N, processor becomes efficient.

3. Performance saturates at some value of N set by L1 cache - all 4 arrays fit into cache

5. The above value depends on bandwidth and latency of L1 cache.

6. Further increase in N -- sharp drop in performance because innermost cache is not large enough to hold all the data.

7. L2 has similar bandwidth that of L1 but higher latency.

8. Now L1 has to support both - provide data to registers as well as continuously evict cache lines from L2.

Lecture 5
HPC: CS301
Course Instructor: B. Chaudhury

Previous Lecture:

Memory Levels : DRAM and Caches

Cache Hit and Cache Miss
Cache Lines: 64 bytes (CPU requests data from RAM in blocks of 64 bytes)
Latency in Clock Cycles.

Clock cyle: 1 tick of the CPUs timer - smallest possible duration of the time for the device. 1GHz clock cycle - 1 billion ticks per second.

Latency - the time the CPU takes to wait to perform something (measured in clock cycles)

X86/ X64 caches: L1, L2, L3

L1 - smallest (KB) and fastest
L2 - medium sized (hundreds of KB to few MB) and medium speed
L3 - largest (tens of MB) but slowest

Some hypothetical representative numbers

Memory	Latency (Clock Cycles)
Register	1
L1	3-4
L2	15-20
L3	50-60
Memory	100-200
Hard Drive	~ Millions

Cache Lines

The bytes in RAM are grouped in a block of 64 each

for i = 1, N

Travel through the array → forward or backward →

we will get the benefit.

Example of locality

Case-1

```
total=0
for (i=0,i<n,i++)
total + = a[i]
}
return total
```

Which locality (data)?
Look into each iteration

Spatial → a[i]
Temporal → total

a[1] a[2] ... a[3]

Skill: identify the pattern of data access in the code/ algorithm

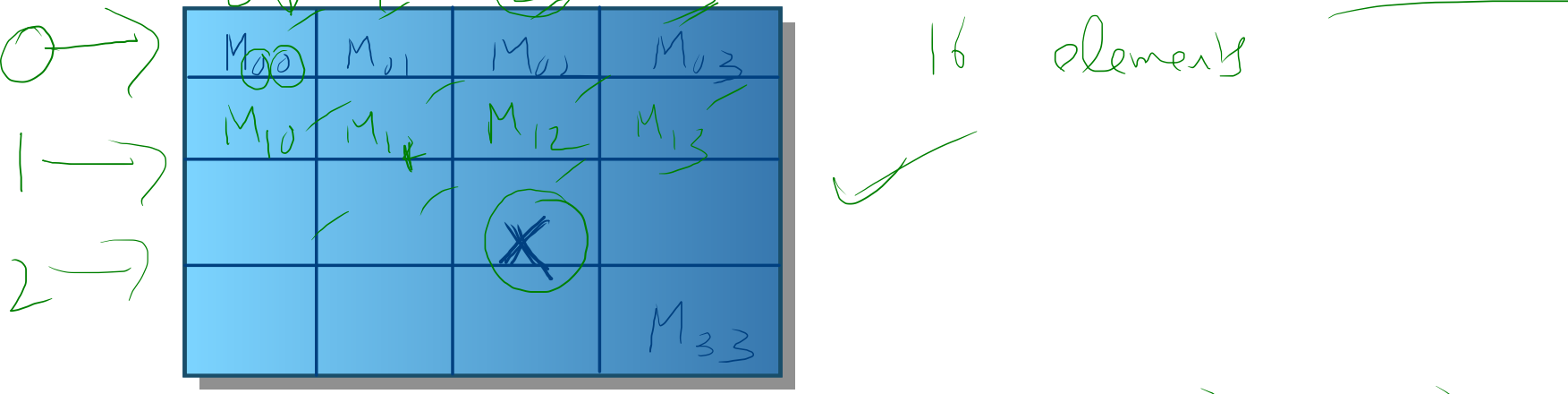
CASE-2 (C code)
n is very large

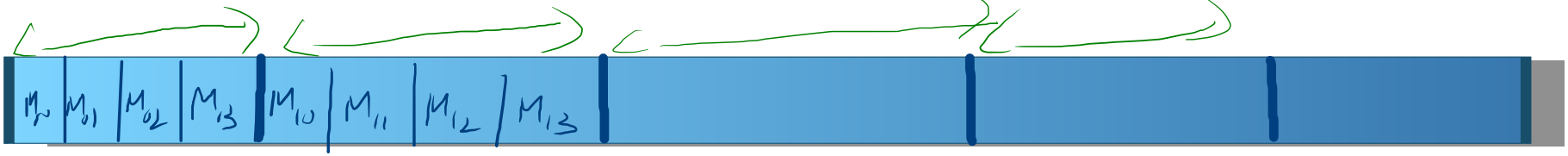
```
total=0
for (i=0,i<n,i++)
        for (j=0, j<n, j++)
                for(k=0, k<n, k++)
total + = a[k] [i] [j]

return total
```

Optimization for the memory hierarchy - improve locality
** Proper choice of algorithms*
** Lop transformations*

Case Study Matrix multiplication

Row major layout for a 2D C array (4 x 4 matrix)





Linearized into 16 element 1D array - flat memory space in modern computers

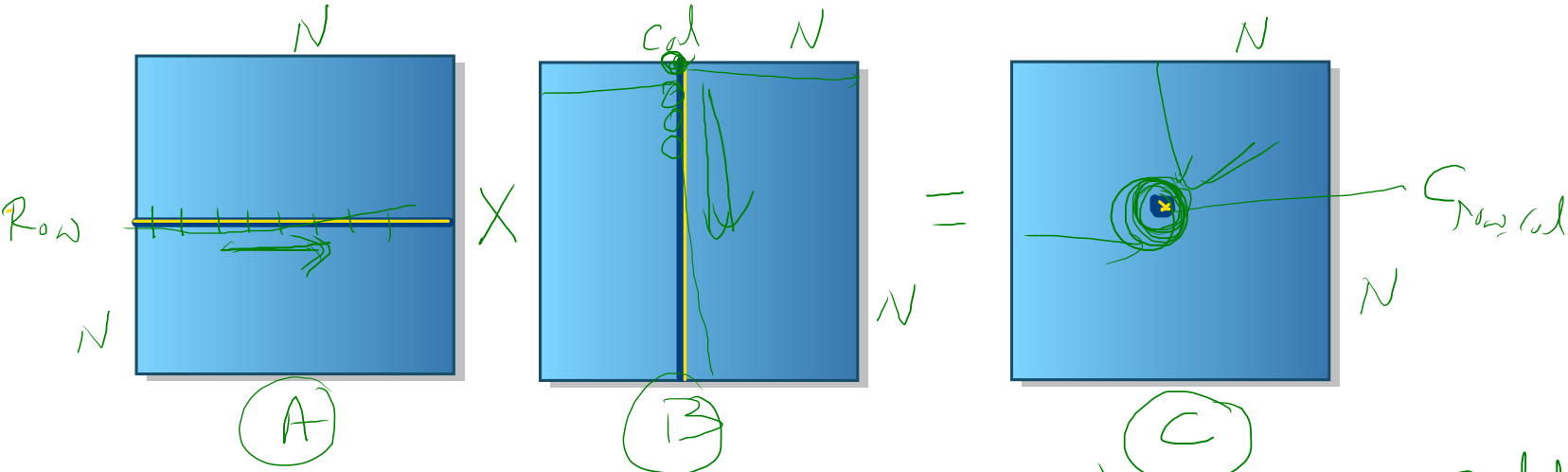
*Element = Row * width + column*

10th element = 2 x 4 + 2 = 10

Matrix multiplication code

```
int row, col, k
for (row=0, row<N, row++)
        for (col=0,col<N, col++ )
            for (k=0, k<N, k++)
element [row*width + col] += A[row*N + k] * B[k*N + col]
```

where N = width of the matrix



Total operations → Computation → $O(2N^3)$

Total memory access → $3N^2$

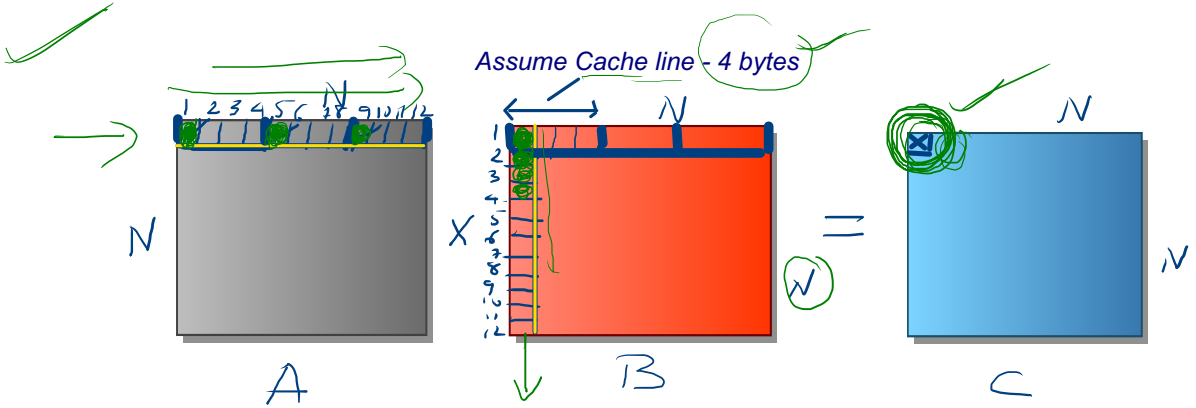
Compute to Memory Access

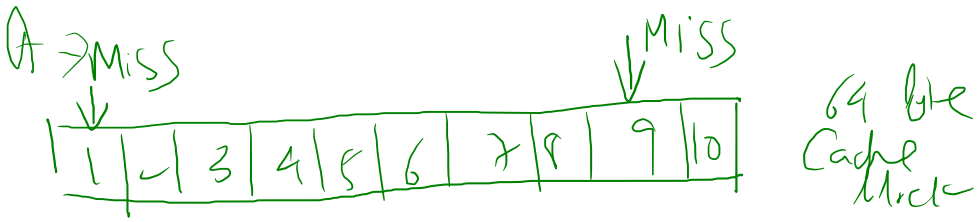
$CMA = \frac{2N^3}{3N^2} = O(N)$

Case Study: Matrix Multiplication

For single element of C (i,j)--> Read : Row (i) x Column (j) {dot product}

Cache starts empty, now look into hit and miss





64 byte Cache block

If matrices are stored in rows in RAM -- reading columns will be slow (not using cache lines)

Cache miss analysis

* *Matrix elements - double (8 bytes)*
* *Cache block - 64 bytes (8 elements)*
* *Cache size C << N*

Lets say N= 10000

First iteration:

Total Cache Misses

$= \frac{N}{8} + N$ (A) (B)

$= \frac{9N}{8}$ misses

Total Misses

$= \frac{9N}{8} \times N^2 = \frac{9N^3}{8}$

Matrix - $10^4 \times 10^4$

A -> 8×10^8 B

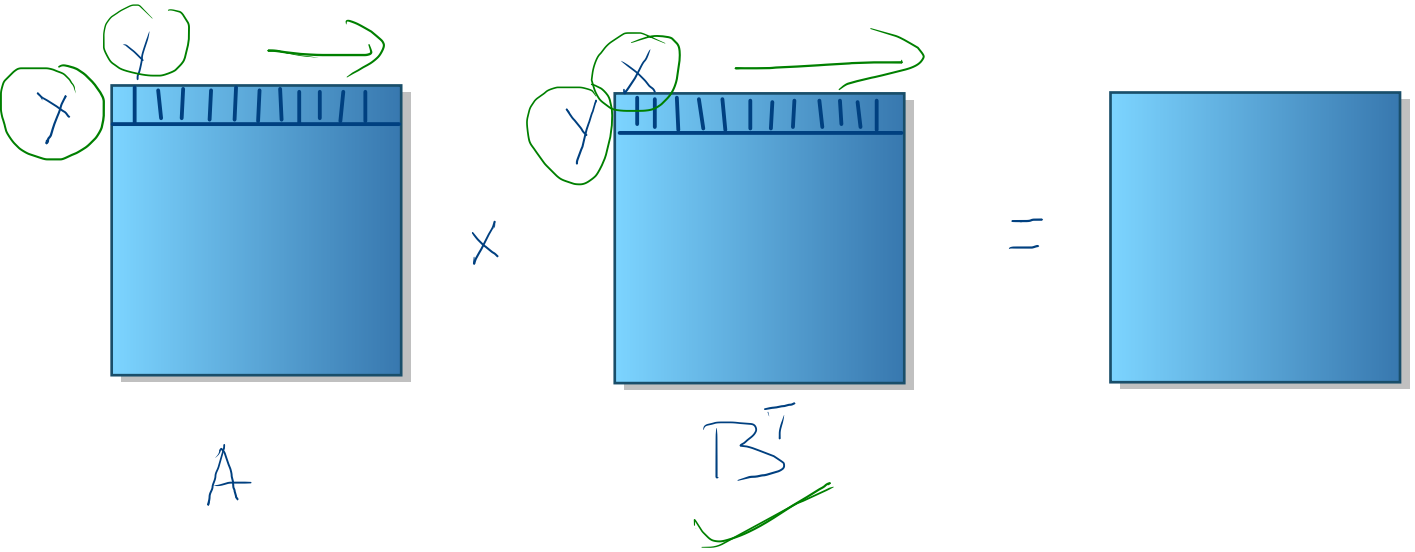
Matrix - $10^2 \times 10^2$

A -> 80 KB

Matrix -> $10^3 \times 10^3$

A -> 8 MB

What if we transpose the matrix B



Read the rows or the columns -- we are using the entire cache lines

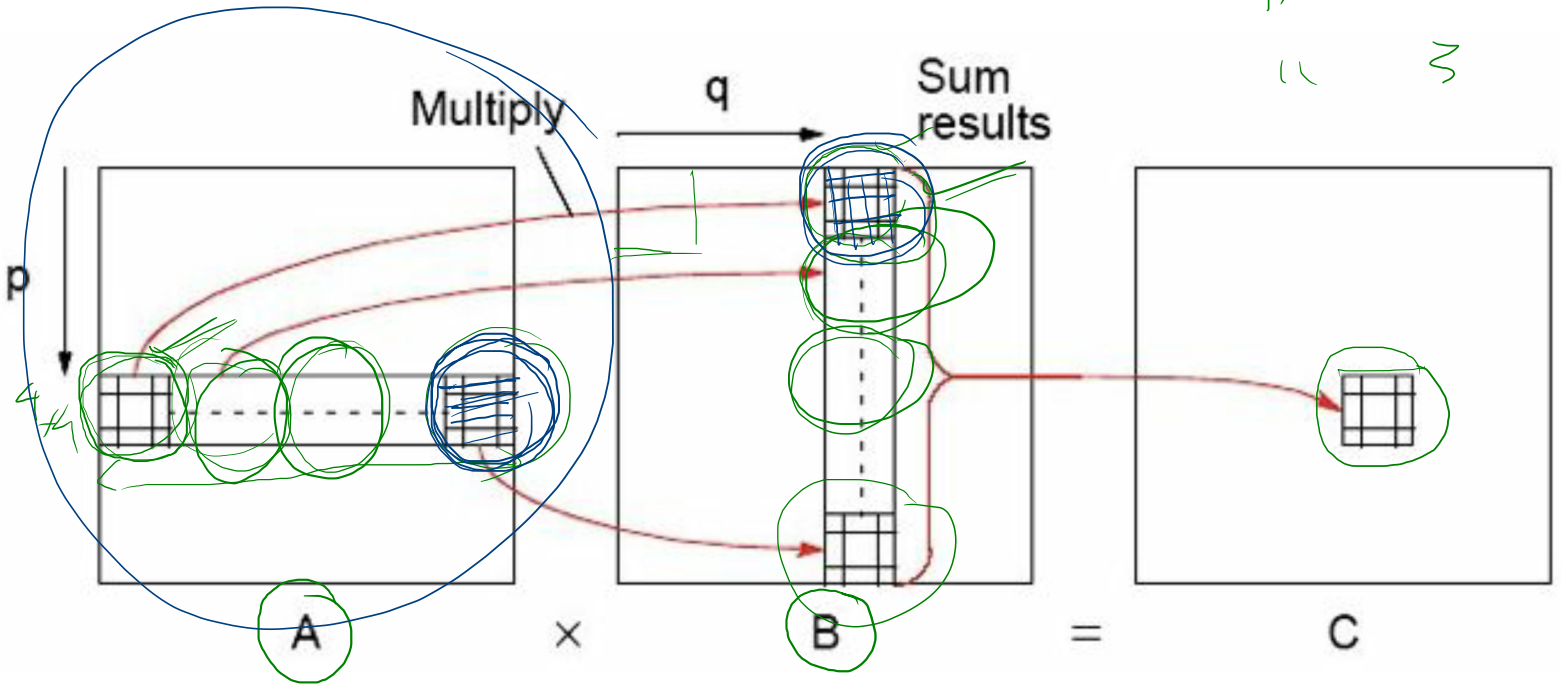
Store the matrices in both in rows and columns -- swap the X and Y corodinates

Message: try to access data in consecutive blocks

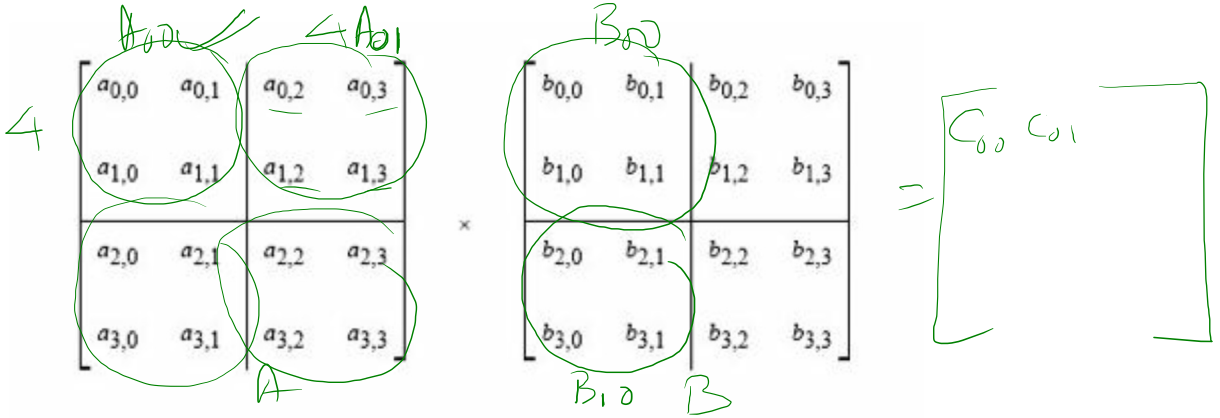
Block matrix multiplication

(Blocking Technique)

Phase 1 calculation



Block matrix multiplication



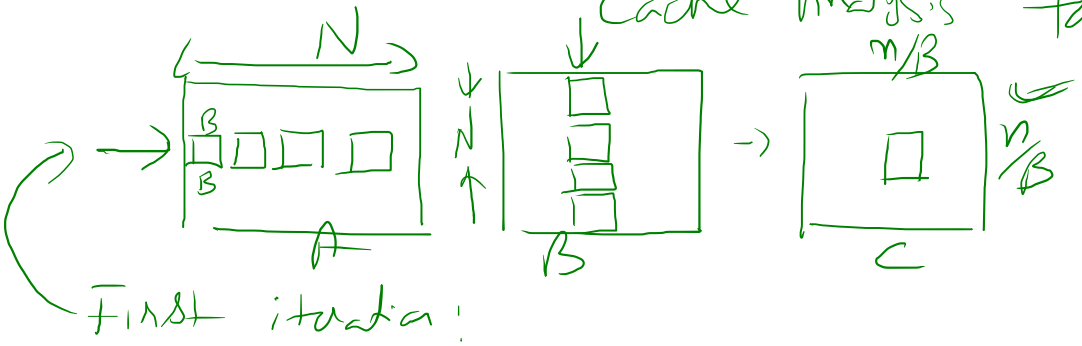
$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix} \times \begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Cache Analysis for Block MM

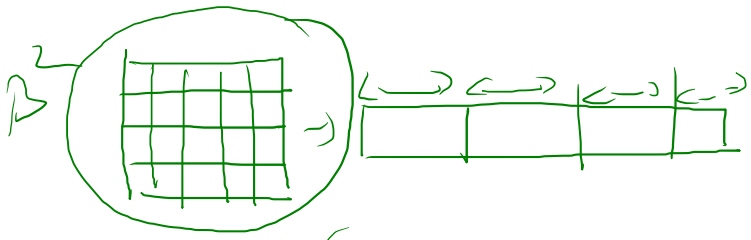


- Cache block → 64 bytes
- Cache Size → $C \ll N$ → Row & Cols size
- Assume ($B \times B$)
- Block size (elements) → B^2

$3B^2 < C$

FIRST iteration:

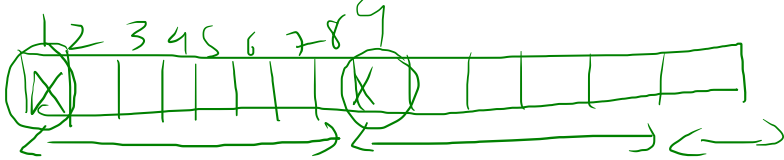
Misses for each block = $\frac{B^2}{8}$ → cache line / size element



How many blocks in vertical & horizontal direction

$$\frac{N}{B} + \frac{N}{B} = \frac{2N}{B}$$

Misses in the first iteration = $\frac{B^2}{8} \times \frac{2N}{B} = \boxed{\frac{NB}{4}}$



Total Misses = $\left(\frac{N}{B}\right)^2 \times \frac{NB}{4} = \boxed{\frac{N^3}{4B}}$

↳ total Blocks

Double Precision vs Single Precision

8 bytes

4 bytes

L1

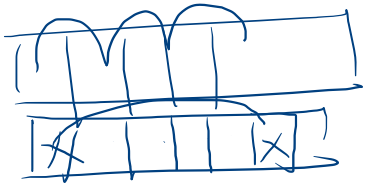
No Blocking → $\frac{9N^3}{8}$

Blocking → $\frac{N^3}{4B}$

$$\text{Gain} = \frac{9N^3/8}{N^3/4B} = \boxed{\frac{9}{2}B}$$

Block $B=8$ $\quad \frac{9}{2} \times B = \frac{9}{2} \times 8 = 36\times$

Block $B=16$ $\quad \frac{9}{2} \times 16 = 72\times$



64 bytes

* Optimize Code performance
  * Data organization
  * Data Access
  * Loop structured
  * Blocking Technique

* keep working set Small → Temporal Locality
* Small Stride → Spatial locality
* Inner loop code → Data Access

Block size → 4×4

Process → Single, Multiple

how many elements per cache line

4 elements, 8 elements

$3B^2 <$ Cache size

$3B^2 \rightarrow 3\times(4+4)\times8$

→

Compute throughput (FLOPS/Sec) ⇒ how to calculate

Throughput of the Memory system →

64 bit DDRAM interface (Double data Rate)
8 channels
1 GHz clock freq

Peak Access Throughput (GB/S)

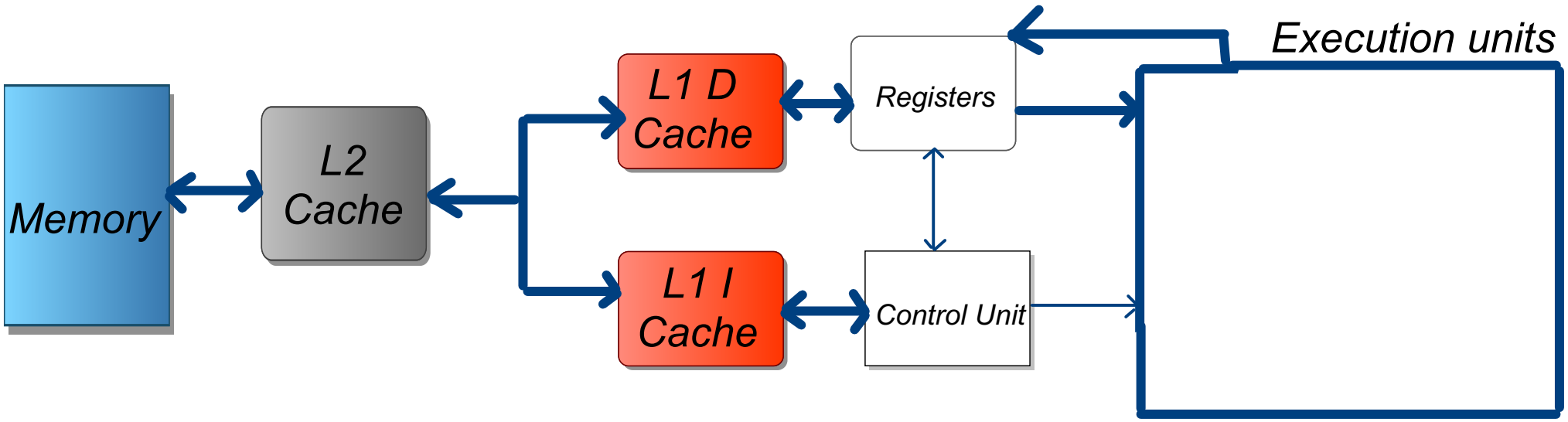
$$= \frac{8\ \text{byte}}{\text{transfer}} \times \frac{2\ \text{transfers}}{\text{Clock channel}} \times 8\ \text{channels} \times 1\text{G}\frac{\text{clocks}}{\text{Sec}}$$

$$= \boxed{\frac{128\text{GB}}{\text{S}}}$$

Each data — 4 byte

Elements/Sec → $\boxed{\frac{32\text{G}}{\text{S}}}$

Memory Hierarchies



Lecture 6
HPC: CS301
Course Instructor: B. Chaudhury

Principle of locality → Spatial locality and temporal locality.

Exploit principle of locality. Memory access on blocks of data instead of individual data items. These are called cache blocks.

When two statements can be executed in parallel?

Suppose we have 2 stetments

statement 1
statement 2

and we wish (2 processors execute indpendently, i.e. no control over order of execution between processor):
statement 1 in proc 1
statement 2 in proc 2

Main requirment: their order of execution should not matter i.e.

statement 2	Equivalent	statement 1
statement 1		statement 2

case 1
a=1
b=2

case 2
a=1
b=a

Case 3
b=a
a=1

Case 4
a=2
a=1

Case 5
a=f(y)
b=a

Dependences : types

** Flow (true) dependence*

statement i precedes j and i computes a value that j uses

(1) x=1 (2) y=x+2 (3)x=z-w (4) x=y/z

1 → 2
2 → 4

** Anti dependence*

statement i precedes j and i uses a value that j computes

(1) x=1 (2) y=x+2 (3)x=z-w (4) x=y/z

2 → 3
2 → 4

** Output dependence*

statement i precedes j and i computes a value that j also computes

(1) x=1 (2) y=x+2 (3)x=z-w (4) x=y/z

1 & 3
3 & 4

** Input dependence*

statement i precedes j and i uses a value that j also uses

(1) x=1 (2) y=x+2 (3)x=z-w (4) x=y/z

3 → 4

* If we say dependence flows from i to j then i is source and j is sink.
Flow dependence is the true dependence, other dependences can be eliminated by renaming

* Data dependency graph: (1) nodes are the statements and (2) directed edges are dependence relations

Dependence between loop iteration

```
for i=1,n
for j=2,m
b[i,j] = ----- + b[i,j-1]
```

* iterations of the loop j must be done sequentially
* " " " " " i can be executed in parallel.

(1) loop independent dependence
(2) loop carried dependence

```
for i=1,N
a[i] = b[i] +c[i]
d[i] = a[i]
```

i=1 — core 1
i=2 → core 2
i=3 → core 3

```
a[1] = b[1] + c[1]
d[1] = a[1]
```

```
a[2] = b[2] + c[2]
d[2] = a[2]
```

in the above case dependence flows within same iteration (loop independent ; dependence distance=0)

```
a[i] = b[i] +c[i]
d[i] = a[i -1]
```

```
a[1] = b[1] + c[1]
d[1] = a[0]
```

```
a[2] = b[2] + c[2]
d[2] = a[1]
```

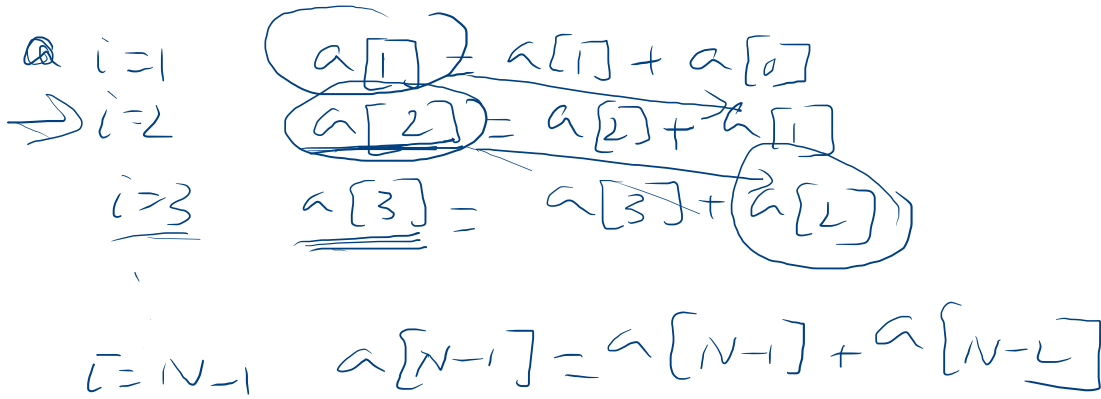
dependence flows from 1 iteration to next (loop carried dependence; dependence distance=1)

Loop carried dependences

```
a[0]=1
for (i=1; i<N; i++) {
a[i]= a[i] +a[i-1]
}
```

Can this loop be parallelized?

Approach:
Look into the pattern for some iterations



How to detect dependences

* Look carefully how each variable is used within a iteration
* Is the variable only read and not written (if yes - no dependences)
* For each variable which is writtten - find out whether there is any accesses in other iterations than the current? if yes then depepnces are present

Other important points for no dependences:
* Each element is assigned by at most one iteration
* No iteration reads elements assigned by any other iteration

```
for (i=1; i<N; i+=2) {
a[i]= a[i] +a[i-1]
}
```

Can this loop be parallelized?

Approach:

Look into the pattern for some iterations

a[1] = a[1] + a[0]

a[3] = a[3] + a[2]

a[5] = a[5] + a[4]

```
for (i=0; i<N/2; i++) {
a[i]= a[i] +a[i+N/2]
}
```

Can this loop be parallelized? Yes

Approach:

Look into the pattern for some iterations

$a[1] = a[1] + a[1 + N/2]$

$a[2] = a[2] + a[2 + N/2]$

$a[N/2 - 1] = a[N/2 - 1] + a[N/2 - 1 + N/2]$

```
for (i=0; i<=N/2; i++) {
a[i]= a[i] +a[i+N/2]
}
```

Can this loop be parallelized? No

Approach:

Look into the pattern for some iterations

$N = 10$

$a[0] = a[0] + a[5]$

$a[5] = a[5] + a[5 + 5]$

$a[N/2] = a[N/2] + a[N/2 + N/2]$

```
a[0]=0
for (i=1; i<N; i++) {
a[i]= a[i-1] +i
}
```

Approach:

$i=1 \quad a[1] = a[0] + 1 = 0+1 = 1$

$i=2 \quad a[2] = a[1] + 2 = 1+2 = 3$

$i=3 \quad a[3] = a[2] + 3 = 3+3 = 6$

$i=4 \quad a[4] = a[3] + 4 = 6+4 = 10$

Can this loop be parallelized?

$a[i] = \frac{i(i+1)}{2}$

$1 \rightarrow 3 \rightarrow 6 \rightarrow 10$

$i=4 \rightarrow \frac{4 \times 5}{2} = 10$

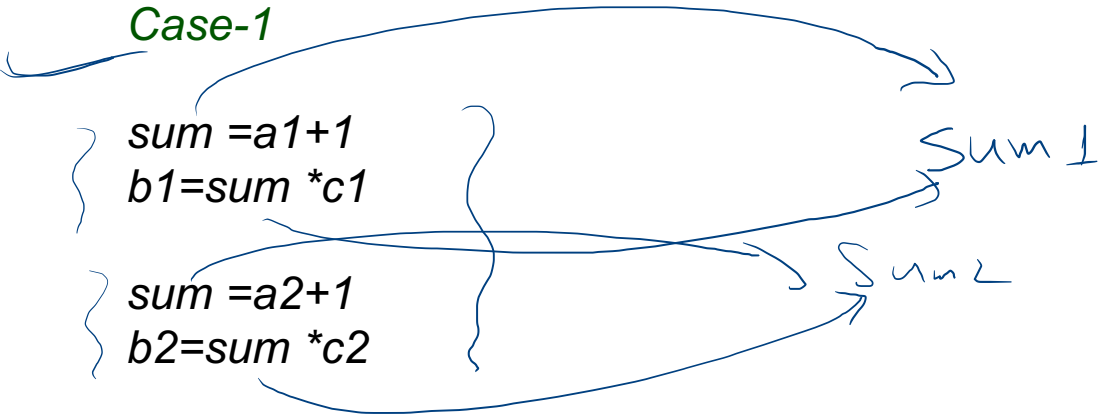
```
for (j=0; j<N; j++) {
a[ id[j] ]= a[ id[j] ] +b[ id[j] ]
}
```

Approach:

Can this loop be parallelized?

Can we remove dependeces in the following cases - is there false dependeces

Case-1



Case 2

```
for (i=0; i<N; i++) {
x=(b[i] + c[i]) /2
a[i]= a[i+1] + x
}
```

Assignment

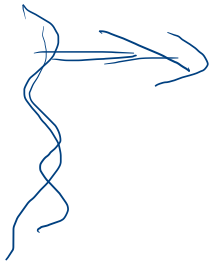
Can this loop be parallelized?

Summary

- *Statement order must not matter.*
- *Statements must not have dependences.*
- *Some dependences can be removed.*
- *Some dependences may not be obvious.*

Eliminating dependence by Scalar Expansion or privatization

```
for (I = 0; I <100, I++
T = A[I];
A[I] = B[I];
B[I] = T;
```



```
T[1] = A[1]
A[1] = B[1]
B[1] = T[1]
```

```
T = A[1]
```

```
T = A[5]
```

Loop interchange & Loop fusion / distribution

- Loop fusion → combining 2 loops into a single loop
  - Reason → Improve locality & (parallelism)
- Loop distribution (opposite of fusion)
  - Reason → Can increase parallelism / Wide locality

→ Can turn a non-parallelizable loop into a parallelizable loop

$N >> 1$

Code 1

```
for (i=0, i<N, i++)
    a[i-1] = b[i]
for (j=0, j<N, j++)
    c[j] = a[j]
```

fusion ⟷

```
for (i=0, i<N, i++)
    a[i-1] = b[i]
    c[i] = a[i]
```

Can this code be parallelized? Yes

Disadvantage → Worse locality

Benefit → Better cache locality

Disadvantage → it cannot be parallelized

Simple Optimizations of serial code

Case A vs. Case B

Case A

```
logical :: flag
flag = .false.
do i=1,N
if(comlex_func(A(i)) < TRESHOLD) then
flag=.true.
EXIT
Endif
```

Which one we should choose !! Why?

Case B

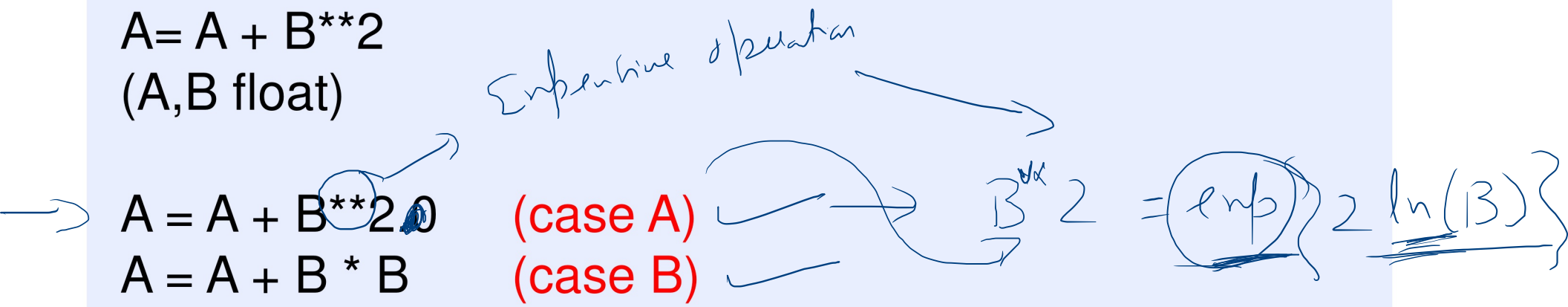
```
logical :: flag
flag = .false.
do i=1,N
if(comlex_func(A(i)) < TRESHOLD) then
flag=.true.
endif
enddo
```

Lesson → Do less work

Case A vs Case B

A= A + B**2
(A,B float)

A = A + B**2.0 (case A)
A = A + B * B (case B)



Strength Reduction

Trigonometric, Exponential

Lesson → Avoid expensive operation

Case A vs B

Case A

```
2  do i=1,N
3     A(i)=A(i)+s+r*sin(x)
4  enddo
```

Case B

```
tmp=s+r*sin(x)
do i=1,N
   A(i)=A(i)+tmp
enddo
```

Case A

```
1  do j=1,N
2    do i=1,N
3      if(i.ge.j) then
4        sign=1.d0
5      else if(i.lt.j) then
6        sign=-1.d0
7      else
8        sign=0.d0
9      endif
10     C(j) = C(j) + sign * A(i,j) * B(i)
11   enddo
12 enddo
```

Case B

```
1  do j=1,N
2    do i=j+1,N
3      C(j) = C(j) + A(i,j) * B(i)
4    enddo
5  enddo
6  do j=1,N
7    do i=1,j-1
8      C(j) = C(j) - A(i,j) * B(i)
9    enddo
10 enddo
```

Case A

```
do i=1,N,2
 do j=1,N
  c(i)=c(i)+a(i,j)     *b(j)
  c(i+1)=c(i+1)+a(j,i+1)*b(j)
 enddo
enddo
```

Loop unrolling

Case B

```
do i=1,N
 do j=1,N
  c(i)=c(i)+a(j,i)*b(j)
 enddo
enddo
```

Optimizations – lessons learned

- **Do less work!**
- **Avoid expensive operations!**

FP MULT & FP ADD are the fastest way to compute

- **Replace expensive functions by table lookup**
  Avoid DIV / SQRT / SIN / COS / TAN → table lookup
- Explore whether data is close to CPU!
- **Elimination of common subexpressions!**
- **Avoid branches!**

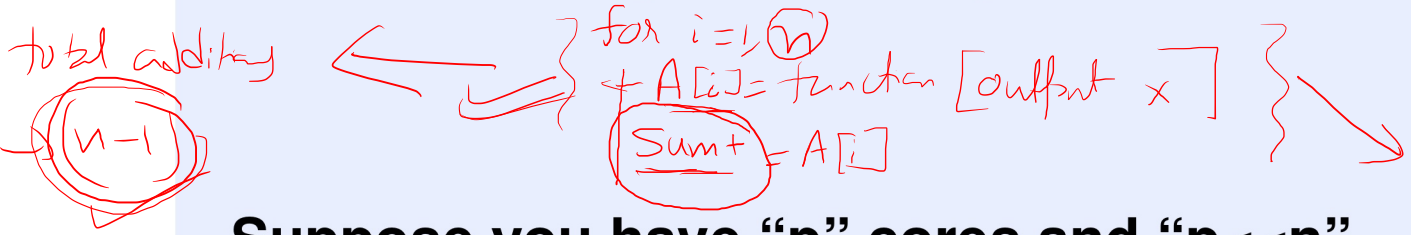
Support the compiler to understand and optimize your code!

Lecture 7
HPC: CS301
Course Instructor: B. Chaudhury

First example *of introducing parallelism using a simple algorithm*

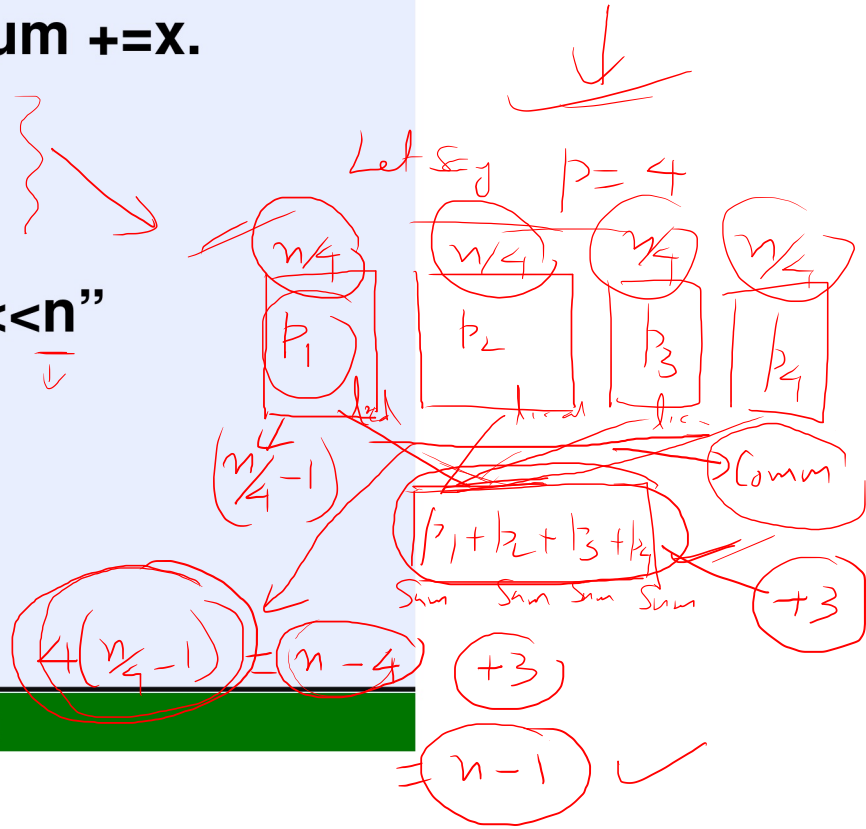
Compute n values and add them together

Serial code – simple !! →
Sum=0; for loop i=1,n; compute x; sum +=x.



Suppose you have "p" cores and "p<<n"

How you will do it in parallel?



First example

Compute n values and add them together

Suppose you have “p” cores and “p<<n”

“partial -sum” for each core

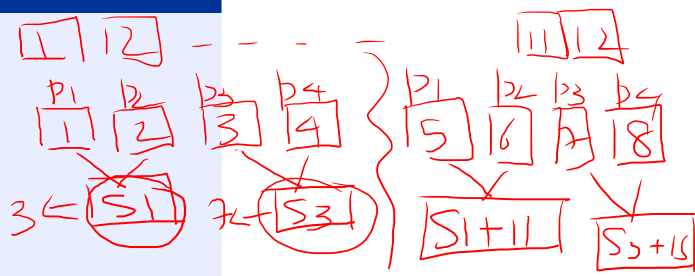
Each core → divide in independent blocks, private variables, Say “my_sum” of each core.

Then “global-sum” → sum of all “my_sum”

Master core gets all “my_sum” from other cores.

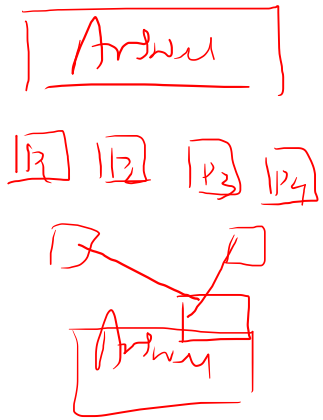
First Example

Is it the best way always?



Method is generalization of the serial global sum ; divide the work among cores and then master core simply repeats the basic serial addition.

<u>Efficient</u> Parallelization is not parallelization of each step of the serial algorithm, but devising an entirely new algorithm.



Granularity

Ratio of computation to communication.

- ***Coarse:*** large amounts of computational work between communication events
- ***Fine:*** small amounts of computational work between communication events

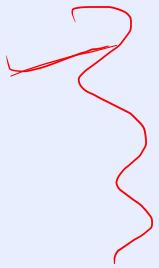
Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

Better algorithm is possible !!

- **Communication among the cores,**
- **load balancing,**
- **and synchronization.**



Complexity of the algorithm.

Types of Parallelism. → Partitioning among cores.

Parallel Computing Metrics - Speedup and efficiency

T(1) - execution time with one core or processor

T(p) - execution time with p processors

Speedup= T(1)/T(p) ---> S(p)

Efficiency E = S(p)/p

Is there any limit on speedup - what if we have infinite processors?

Ideal world S=p, assuming perfect scaling p times with p cores and work is fully parallelizable and constant workload

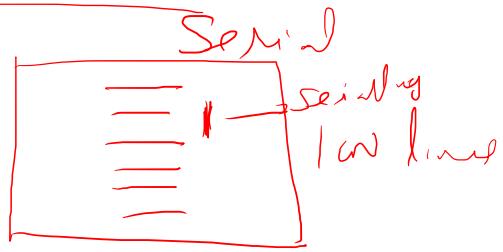
Speedup and efficinecy are not performance (work per unit time) metrics!

Lets assume normalized executioin time for single core can be divided into two parts

T (1) = s+p =1

$p = 1 - s$

- *s- sequential or serial part*
- *p - purely parallel part*



Parallel execution time with N processors T (N)= $s + \frac{p}{N}$

Speedup *S = 1/ (s + (1-s)/N)*

$= \frac{\text{Serial}}{\text{Parallel}} = \frac{1}{s + \frac{(1-s)}{N}}$

$= \frac{1}{S} = \frac{1}{1-p}$

Possibility of parallelization

Amdahl's Law

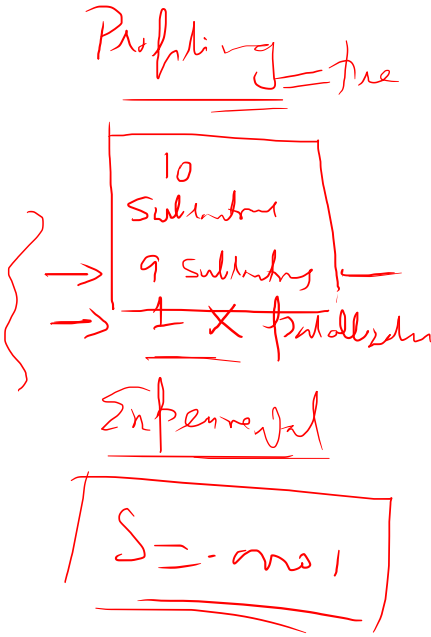
- Amdahl's Law states that potential program speedup → the fraction of code (P) that can be parallelized:

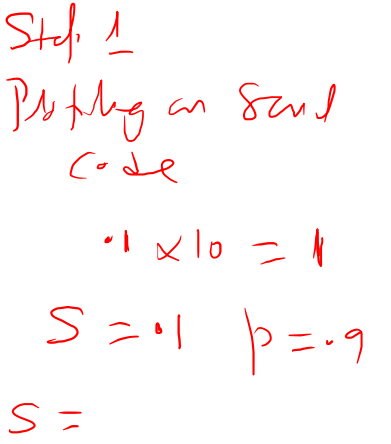
$$\textbf{Speedup} = \frac{1}{1-p}$$

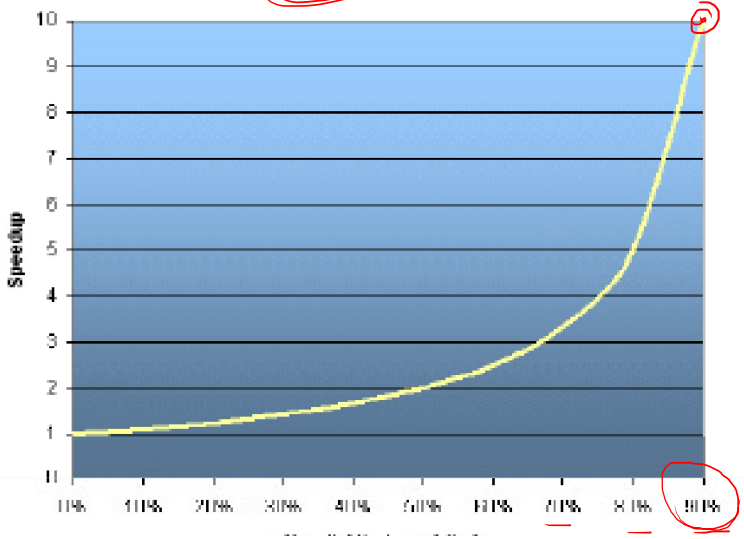
- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).

- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

- Theoretical Maximum speedup ??







Parallel portion of code (in %)

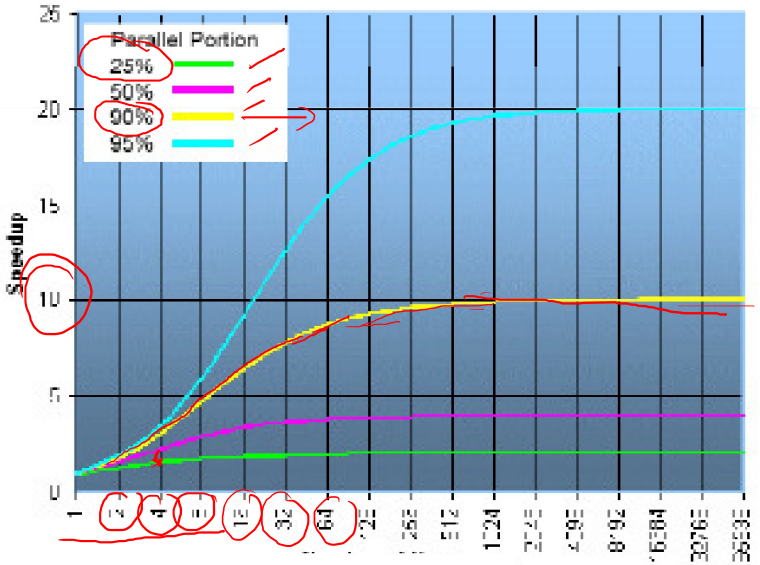
Speedup=1/(1-p)

P in fraction
P+S=1

Y axis → speedup

Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

Speedup=1/[(P/N)+S]



No. of Procecssor (power of 2)

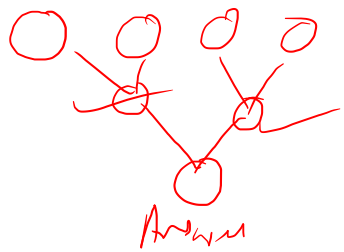
Now if we introduce overhead ?

Two info Variables — n (Problem size), p (no. of Processors)

(a) * Sequential computation time $\sigma(n)$

(b) * Parallel computation time (Serial that can be executed parallelly) $\rightarrow \phi(n)$

(c) Parallel overhead $\rightarrow k(n,p)$



Sequential on a Single processor $= \sigma(n) + \phi(n)$

Parallel time for the parallel code on p processors $= \sigma(n) + \frac{\phi(n)}{p} + k(n,p)$

$$\text{Speedup } \psi(n,p) \left(=, \leq\right) \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + k(n,p)} = \frac{\text{Serial}}{\text{Parallel}}.$$

↳ Not perfectly divided speedup will be less.

$|n|$ 3

33,333

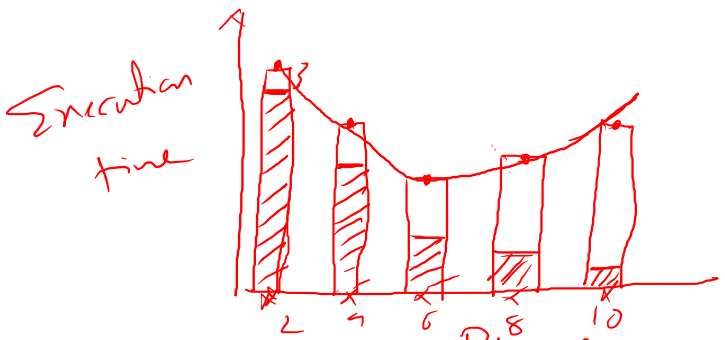
* What will happen if we increase p -

- Computation time decreases
- Communication time increase

$$\psi(n,p) = \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n,p)}$$

Comp, Comp, Comm

fix n → fixing the problem size



Conclusion:→ For any fixed problem size, there is an optimal no of processors that minimizes overall execution time.

Efficiency $= \frac{\text{Speedup}}{P}$ $\quad E = 50\%$ $\quad E = 1\%$

Range of $\epsilon(n,p)$ $\quad$ Range $0 \le \epsilon \le 1$

$$\epsilon(n,p) = \frac{\sigma(n) + \varphi(n)}{p\left[\sigma(n) + \frac{\varphi(n)}{p} + K(n,p)\right]} = \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + pK(n,p)}$$

Computation $\quad$ overhead

Best case Scenario

for parallelization $\rightarrow$ $\sigma(n) \to 0$, $K(n,p) \to 0$ $\quad \epsilon(n,p) \to 1$

Worst case Scenario

$\varphi(n) \to 0$, $\quad \sigma(n) \to$ finite $\quad K(n,p) \to$ finite $\quad p \to \infty$

$\epsilon(n,p) \to 0$

$$\psi(n,p) \leq \frac{\sigma(n)+q(n)}{\sigma(n)+\frac{q(n)}{p}} + k(n,p)$$

$$\psi(n,p) \ll \frac{\sigma(n)+q(n)}{\sigma(n)+\frac{q(n)}{p}}$$

Amdahl's law does not count overhead

Let f denote the inherently sequential portion of the computation

$$f = \frac{\sigma(n)}{\sigma(n)+q(n)} \mapsto f\sigma(n) - \sigma(n) = -fq(n)$$

$$q(n) = \sigma(n)\left[\frac{1-f}{f}\right] = \sigma(n)\left(\frac{1}{f}-1\right)$$

$$\psi(n,p) \leq \frac{\sigma(n)+q(n)}{\sigma(n)+\frac{q(n)}{p}}$$

$$\leq \frac{\sigma(n)+\sigma(n)\left(\frac{1}{f}-1\right)}{\sigma(n)+\frac{\sigma(n)\left(\frac{1}{f}-1\right)}{p}}$$

$$\leq \frac{1}{f+\frac{(1-f)}{p}}$$

$\Rightarrow$ no of processors

Matrix Multiplication Data Structure (Memory Required)

L1 - 32
L2 - 256

N × N

64 × 64 → A, B, C →

Byte

Total = Precision Double

64 × 64 × 8 × (3) = $2^6 \, 2^6 \, 2^3 \times 3$

Per Matrix

$= 2^{15} \times 3$

$= 32\,KB \times 3$

$= 100\,KB$

128 × 128 → 100 KB × (4)

SL = SL

40 KB

100 KB (2^8)

2048 × 2048

$2^7 \times 2^8 = (2^{25})$

$2^{10} \rightarrow 1\,KB$

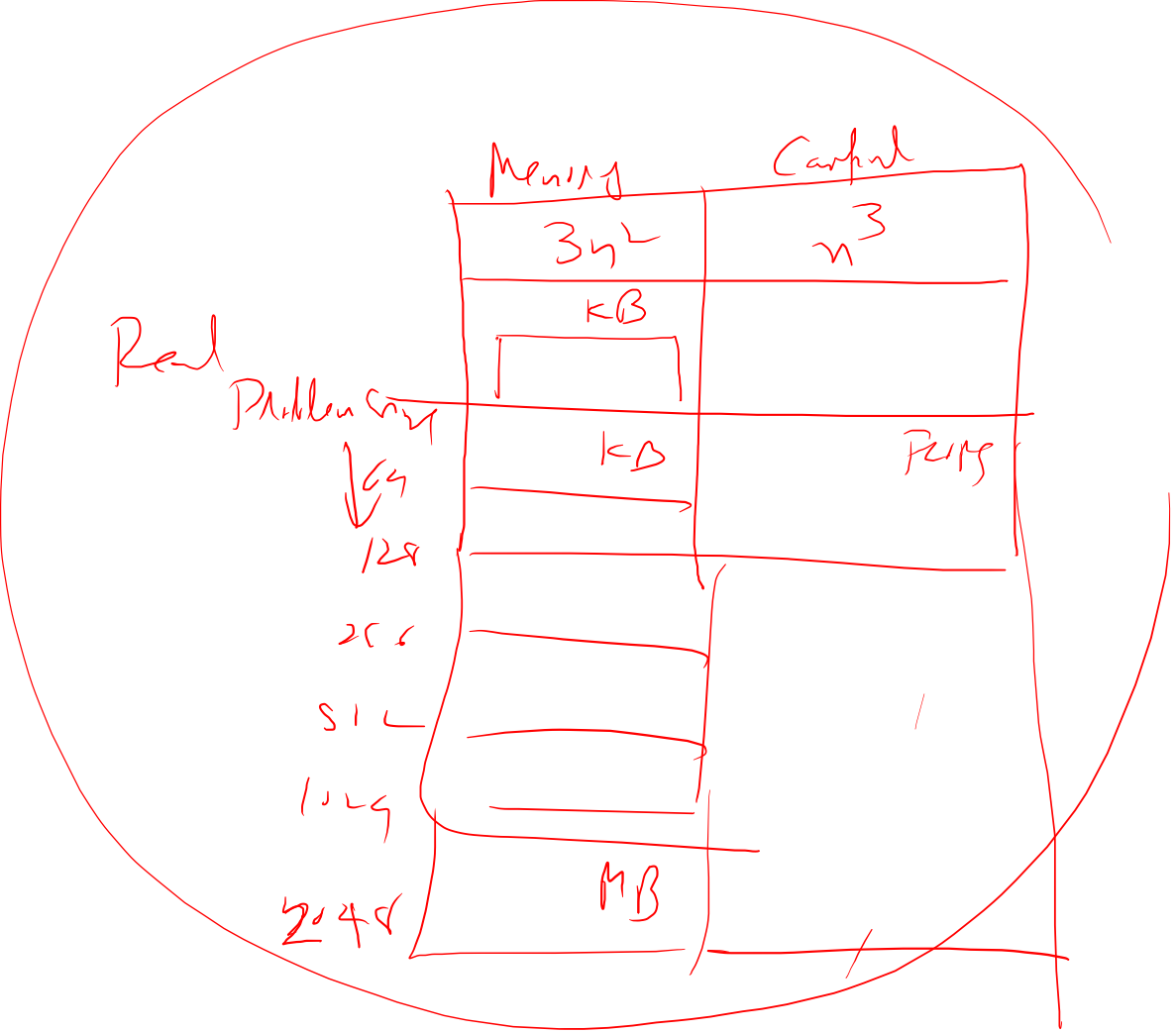
2^{20} MB

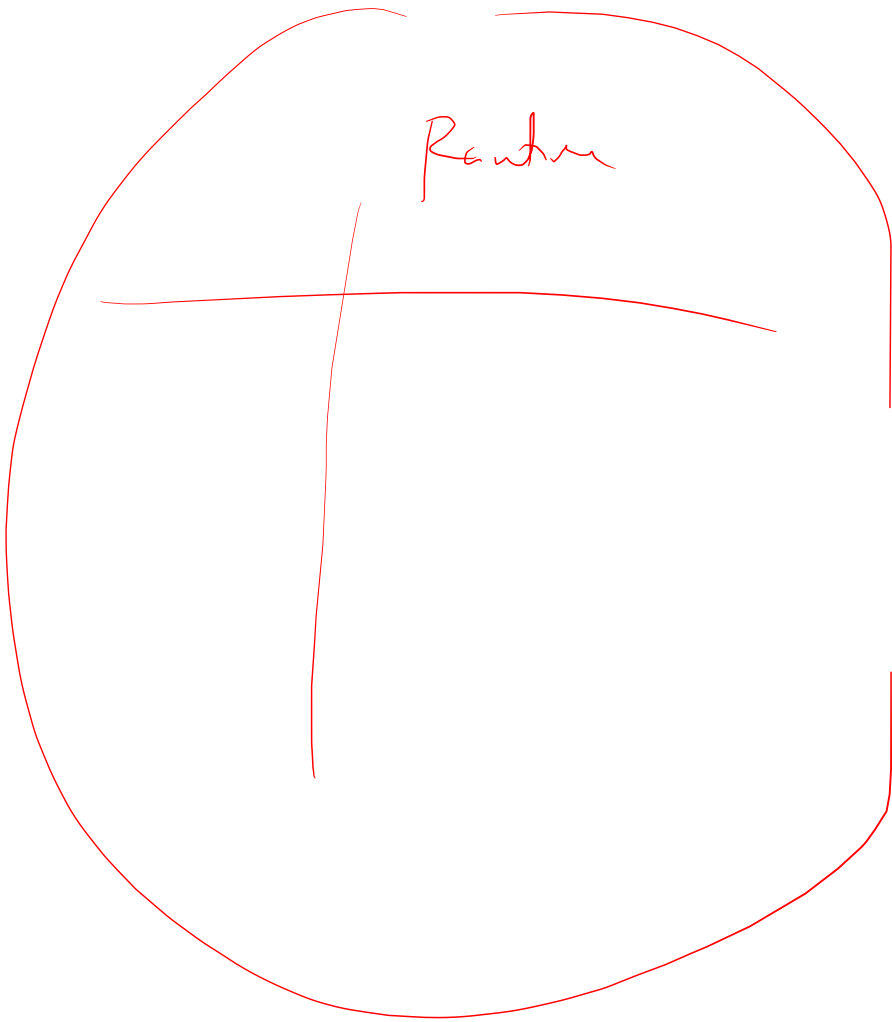
100's KB → Few of MB

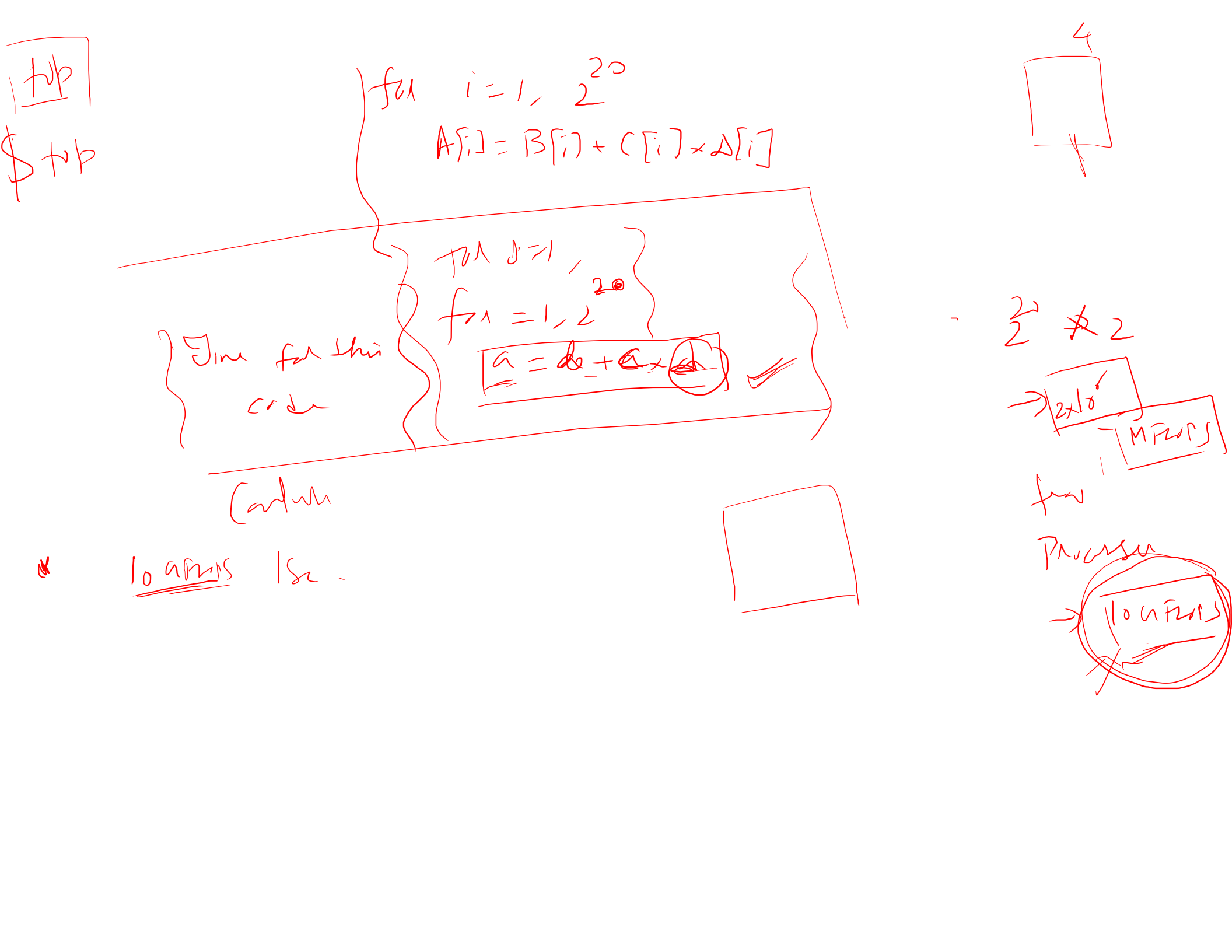
KB → MB

L2 → 256

L3 → 6 MB


Memory
Control
$3n^2$
n^3
KB
Real
Problem Size
KB
Perf
64
128
256
512
1024
MB


Random


tup
$tup
for i = 1, 2^20
A[i] = B[i] + C[i] * D[i]
for j = 1,
for i = 1, 2^20
a = b + c
Time for this code
Cache
10 cycles
4
2^20 * 2
2x10^6
MFLOPS
two
Processor
10 GFLOPS

- Which time ? Resolution → 1 Sec — µs, ms
  Start time
- for i=1, N

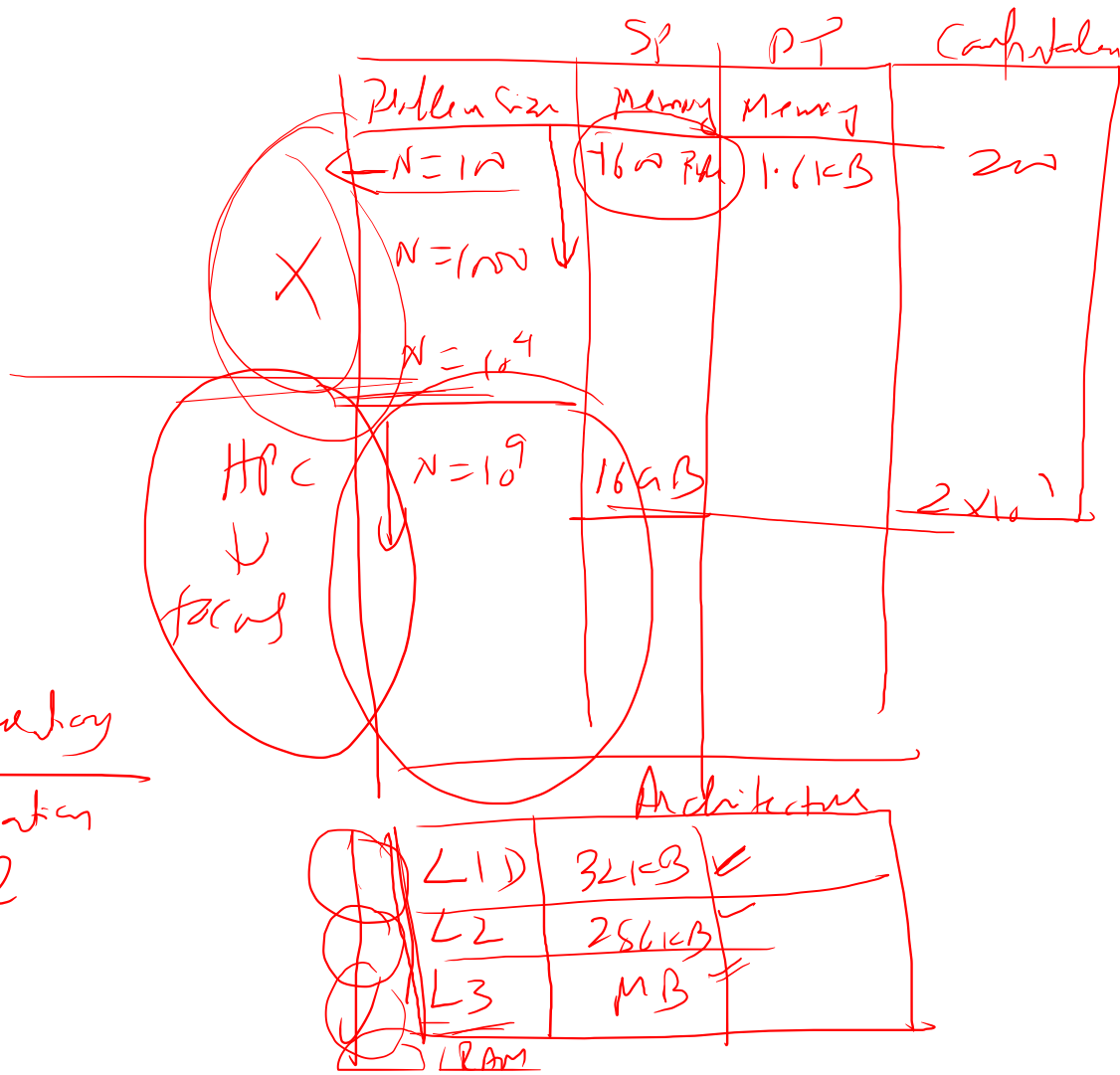
  a[i] = b[i] + c[i] + d[i]

4 × (4 bytes) × N

a, b, c, d — Variables data type — Problem size

end time

Compute → Problem size × No. of operations / iteration × 2



N = 1 M

Compute → 2 M

Data – 10 KB

Compute | Process

10 GFLOPS/Sec

$$\frac{2M}{10 \times 10^9} \text{ seconds}$$

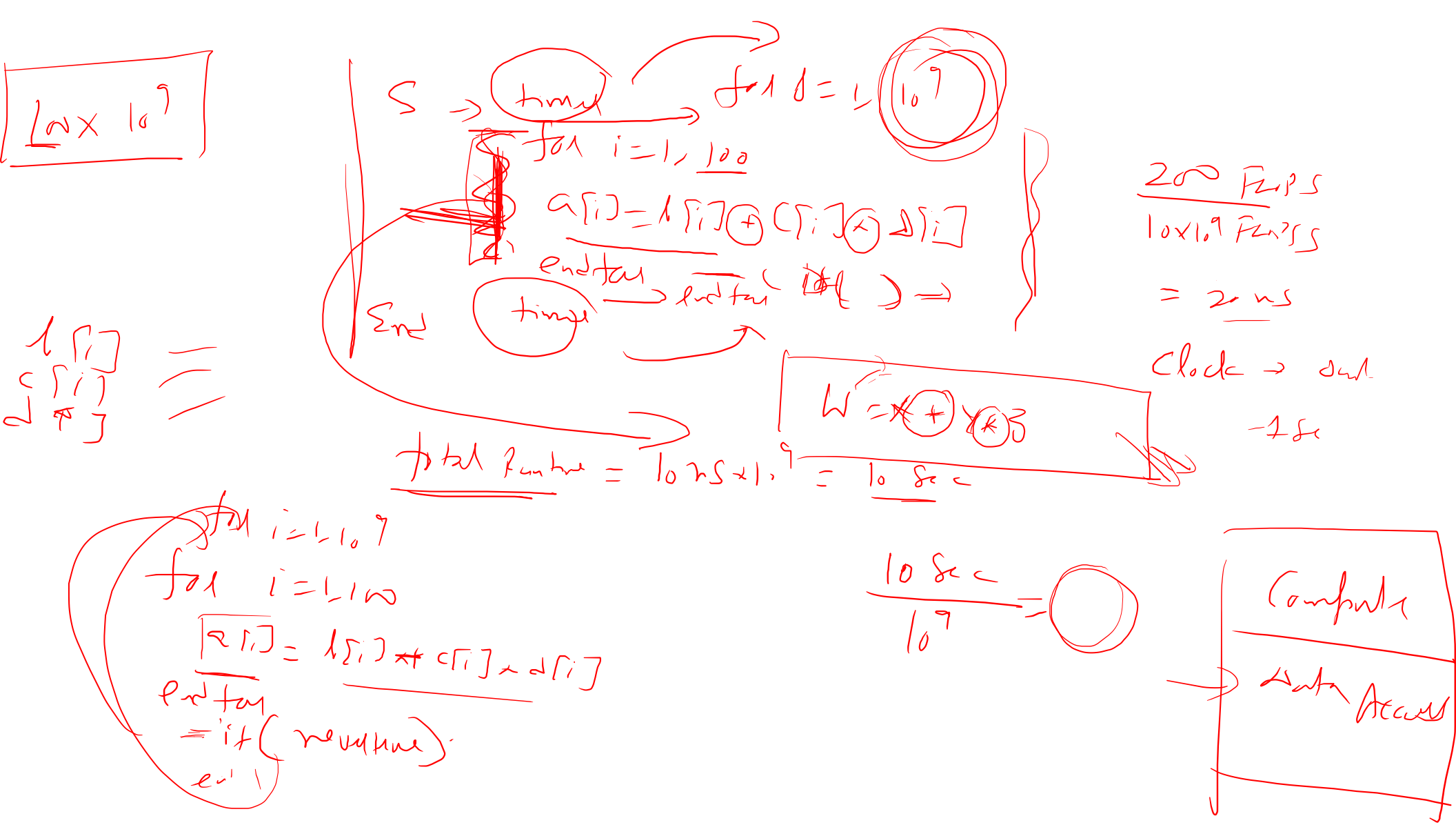
= 2×10 | 200 ns

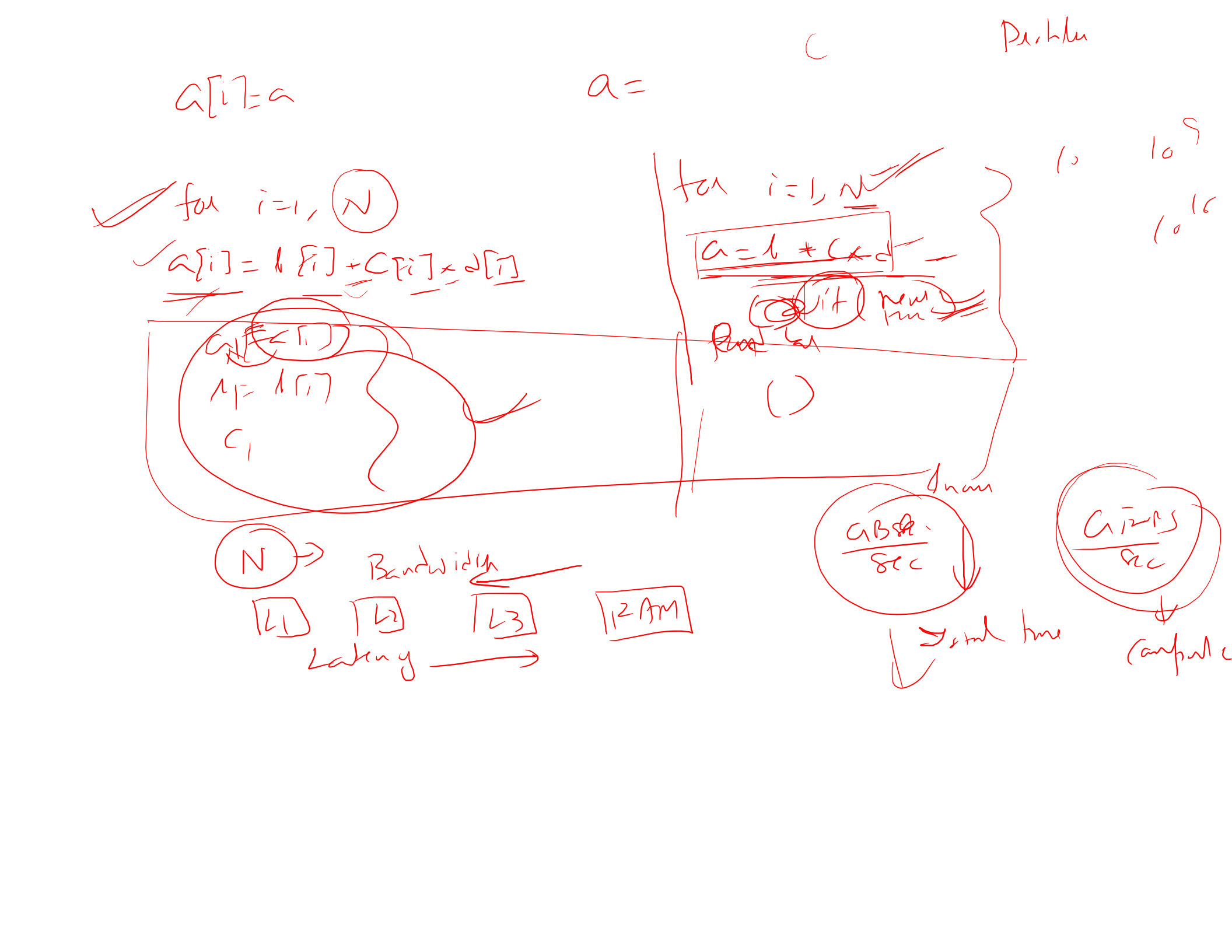
for j = 1, 10^9

for i = 1, N

End for

$$\frac{\text{Total time}}{10^9} = \text{ms}$$


1 ns x 10^9
S
timer
for i=1, 100
a[i]=b[i]+c[i]*d[i]
endfor
timer
for i=1, 10^9
End
20 ns
10x10^9 ns
= 2 s
Clock → 1 ns
W
total Runtime = 10 ns x 10^9
= 10 sec
for i=1, 10^9
for i=1, 100
R[i] = b[i] * c[i] + d[i]
endfor
= if(resultme)
end
10 sec / 10^9
Computer
Data Access


a[i]=a
a=
for i=1, N
a[i] = b[i] + c[i] * d[i]
for i=1, N
a = b + c * d
if
N
Bandwidth
L1
L2
L3
12 GB/s
Latency
GB/s
GB/s

Parallel Programming Complexity

Algorithm +

- Scope of parallelism? (Amdahl's Law)
- Granularity
- Locality
- Load balance
- Coordination and Synchronization

These makes parallel programming harder than sequential programming.