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

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

Performance → Rate → 1/time
- Compute → FLOPS/Sec
- Data Access → Bytes/Sec

System → Non-deterministic

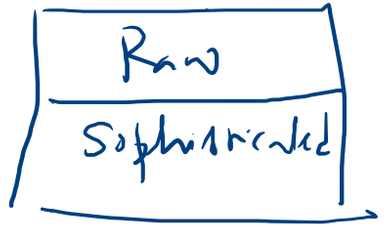
Problem → Algorithm (pattern)
- Data access
- Compute

Car
A — 150 km — B
Theoretically – 1 hr
Reality → 2 hrs

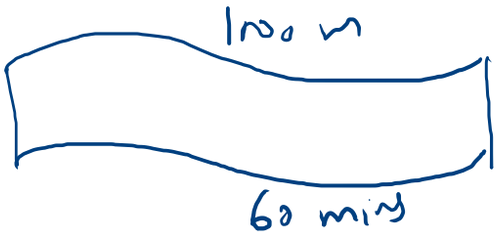
Software → Environment
Algorithm → Road condition
System ← Car
Programmer ← Driver

- What is expected from a good programmer?
  - Accuracy ✓ — Accuracy ∝ No. of Computations
  - Efficiency ✓ — Complexity O(n), O(n³)

Evaluation of the program in the given software-hardware setting



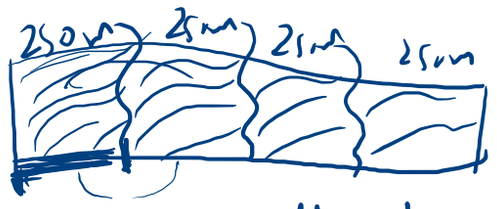
Thinking in serial



- Resource → cleaning Equip
- Workers - processors

Thinking in Parallel

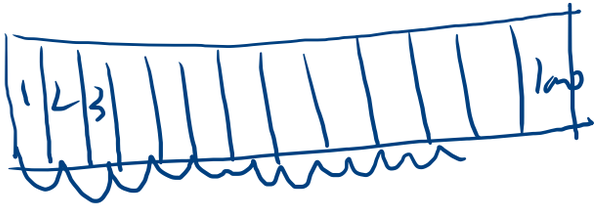
Time - 15 mins



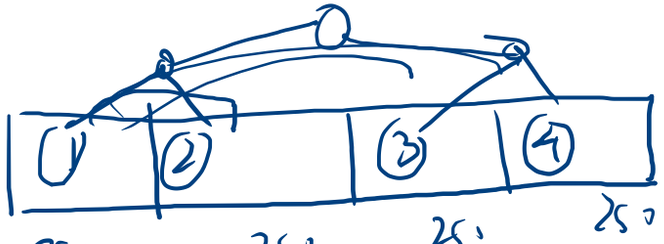
- 4 Equally Efficient workers
- 4 (multiple resources)
- Work Allocation → Load balancing → Decomposition →
- Coordination

Management

Add 1000 numbers



60 mins



250 N's

250 N's

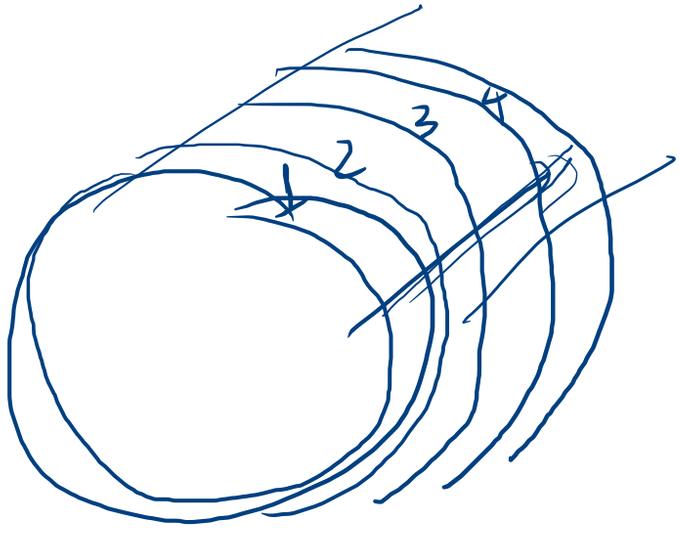
Partial Result

Partial Result

PR

PR

Dependency Analysis

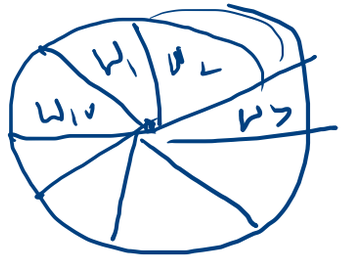


1 m in diameter

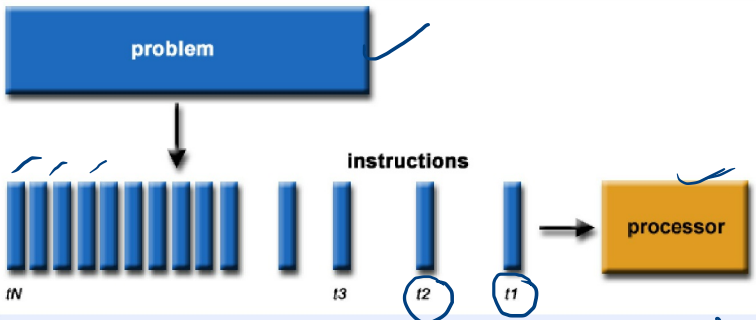
1 m in depth

Serial → 1 Worker → 1 Log - 2nd logs → 3rd logs

Parallel → (10 workers) →



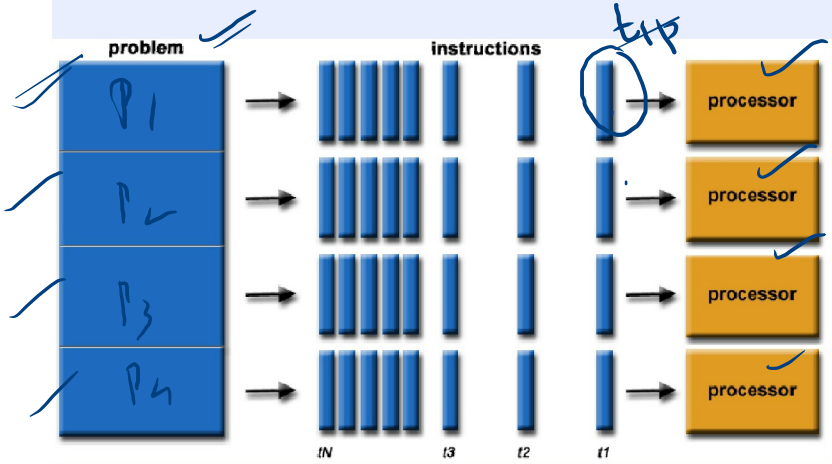
Serial vs Parallel



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.

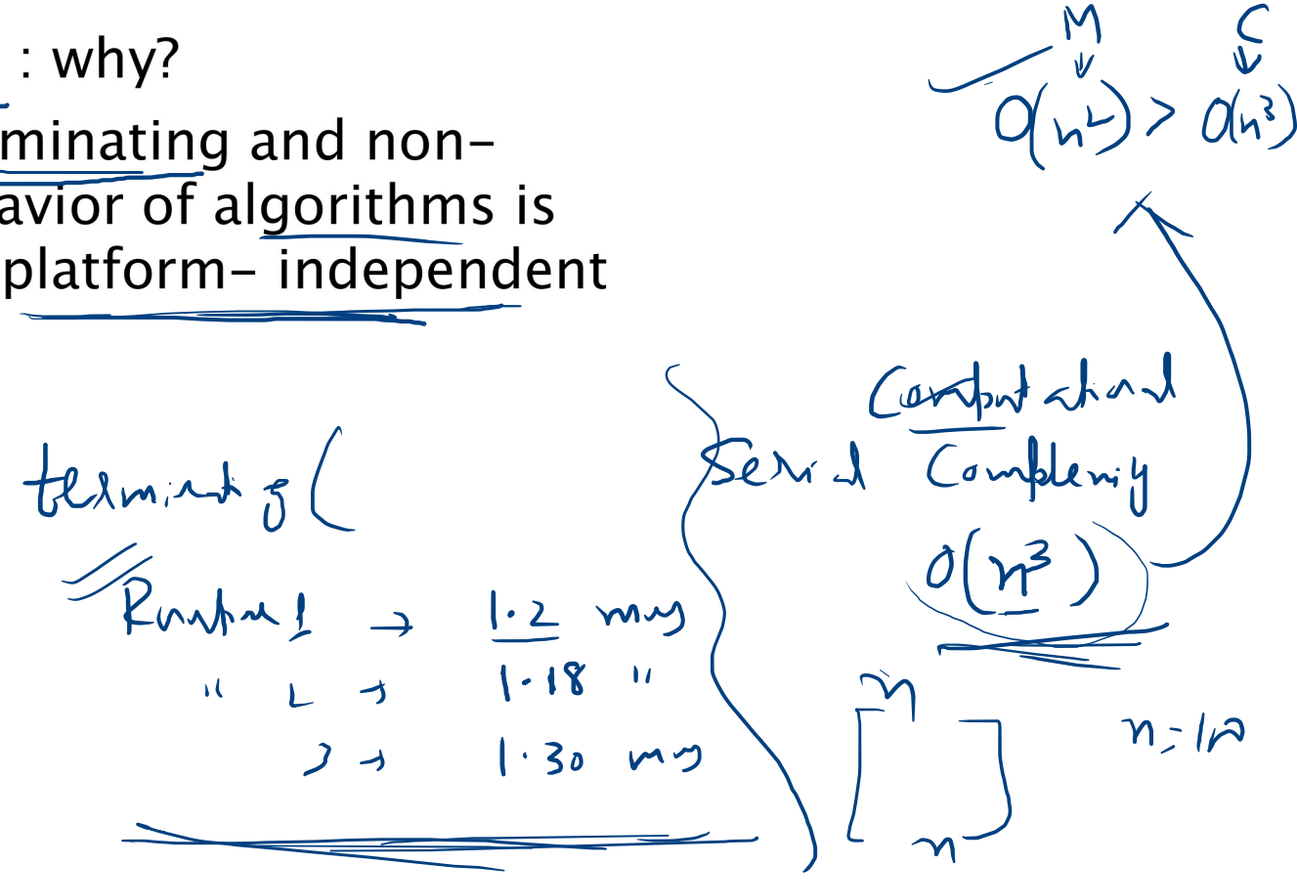
Improve throughput.

$n \times n =$

Viewpoint

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



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

Clock

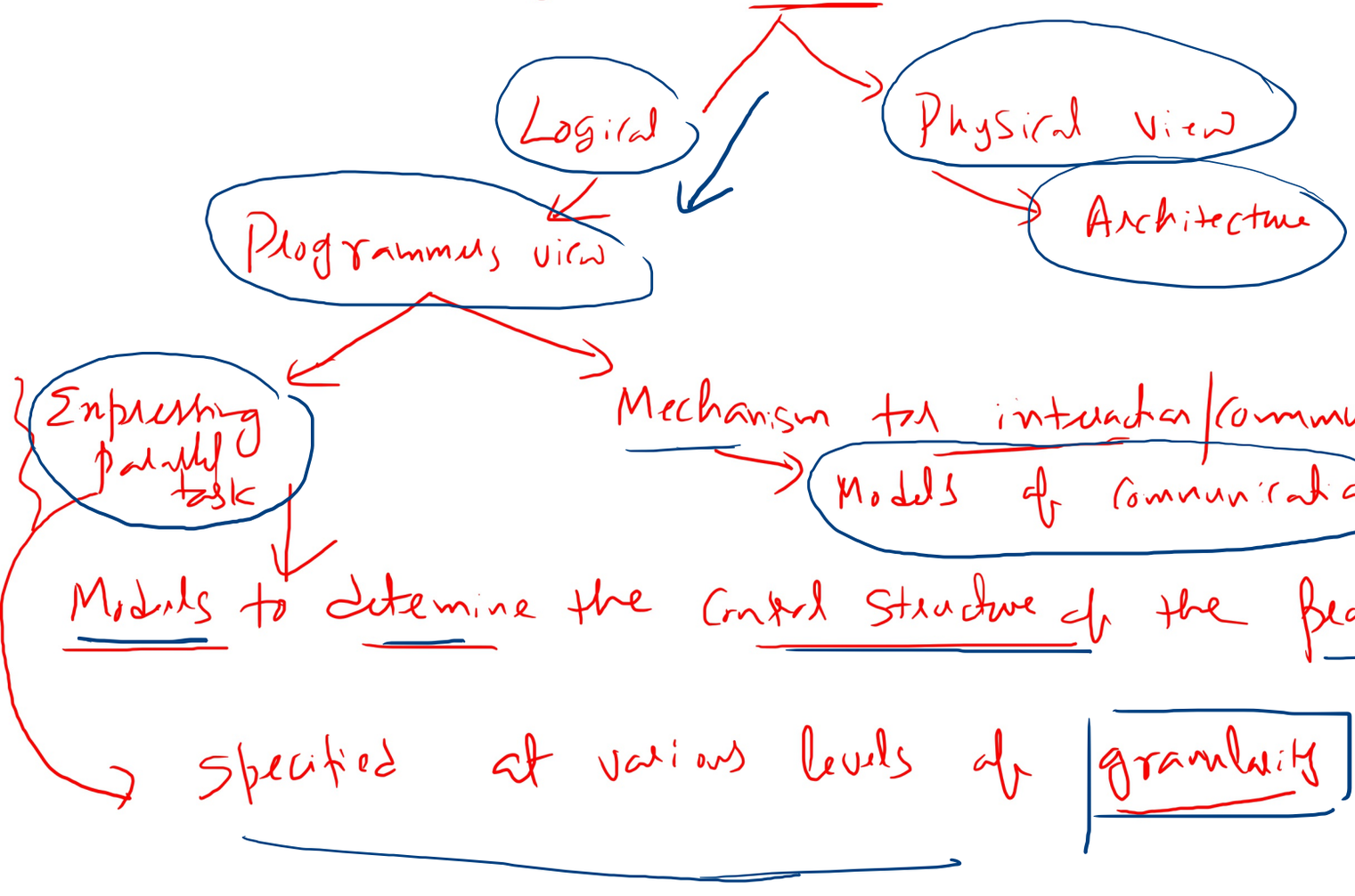
Not enough just to get some speedup, but being able to explain the speedup from a systems viewpoint in terms of resources.

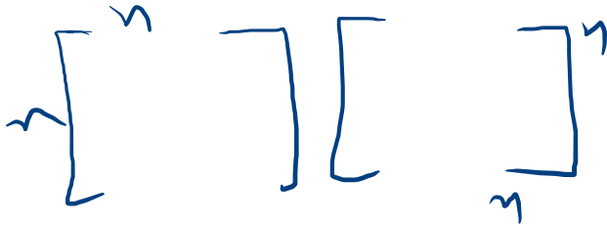
$$\text{Speedup}(S) = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{2}{1} = 2$$

$S = 3$ on $P = 4$

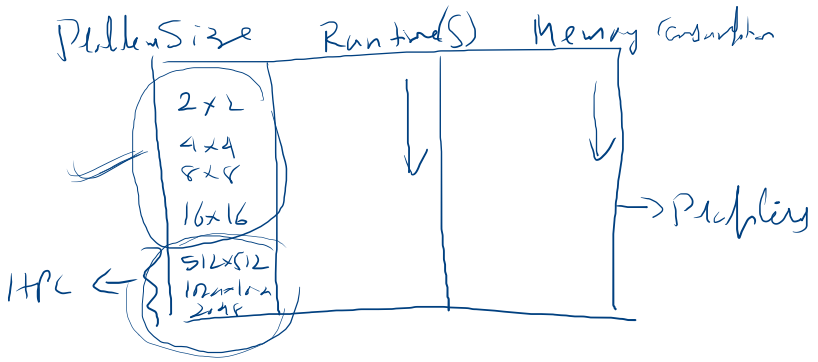
$S = 4.2$ on $P = 4$

Organization of Parallel Platforms

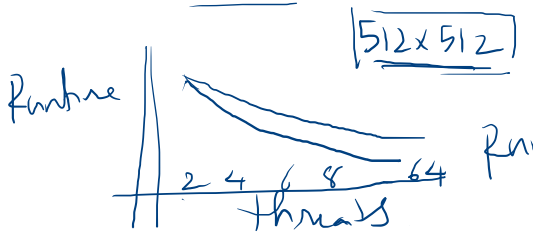


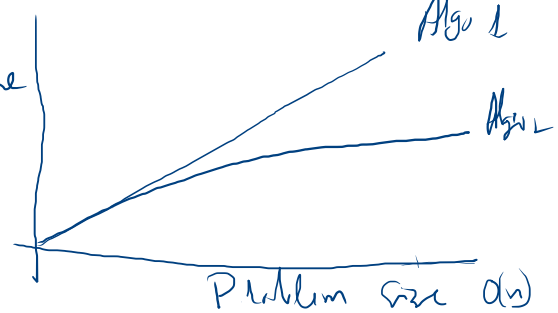


Main Outcome



1. Start with any Serial Algorithm
2. Understand the Data Access and Compute Pattern
3. Implement the most optimized version (multiple versions possible) - right choice of data structures etc.
4. Do runtime analysis - memory requirement and execution time
5. Figure out Scope of Parallelism - Dependency analysis etc.
6. Naive parallel algorithm for shared memory system (multiple threads) - parallel run-time analysis
7. Do profiling and optimize the parallel version - develop a better parallel algorithm - if possible
8. Scalability study
9. Comparision of different parallel versions, speedup, efficiency - as a function of problem-size and architecture
10. Interpretation of run-time data in terms of memory access etc. Comparision with theoretical estimations.





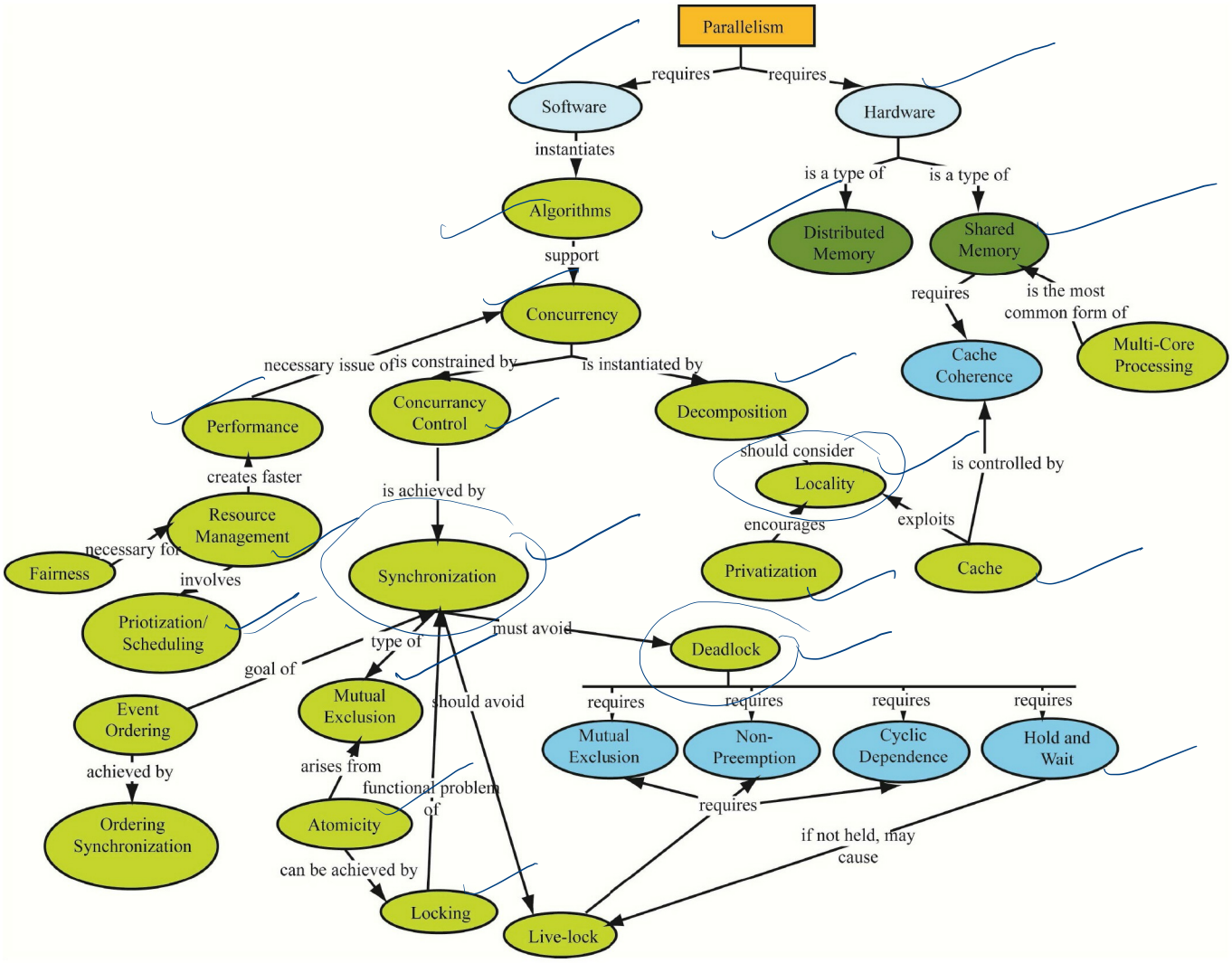


Figure 1. Concept map for parallel computing. Adapted from [9].

Course content

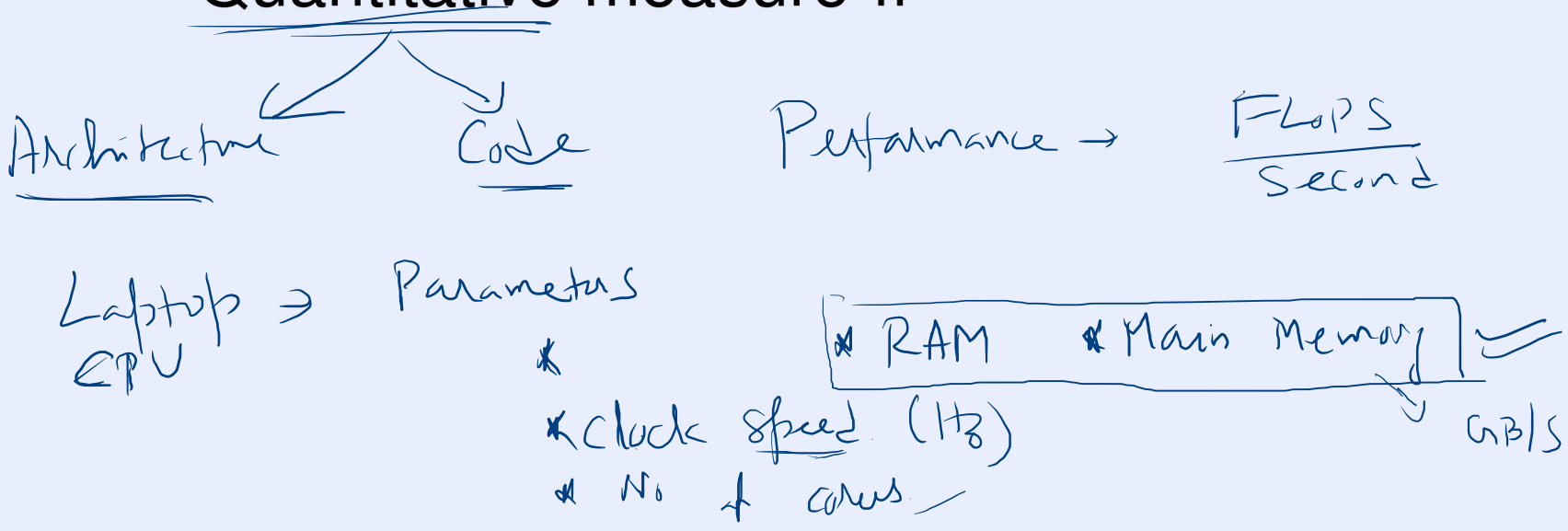
Module 1:

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.

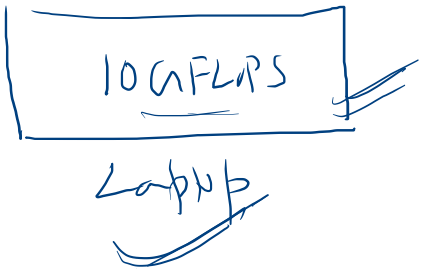
[redacted]

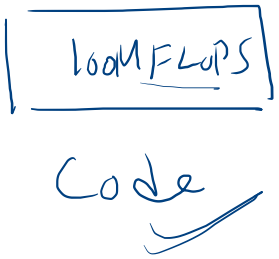
Performance= Work/Time = FLOP/Wall clock time

FLOP (addition, multiplication, division etc.)

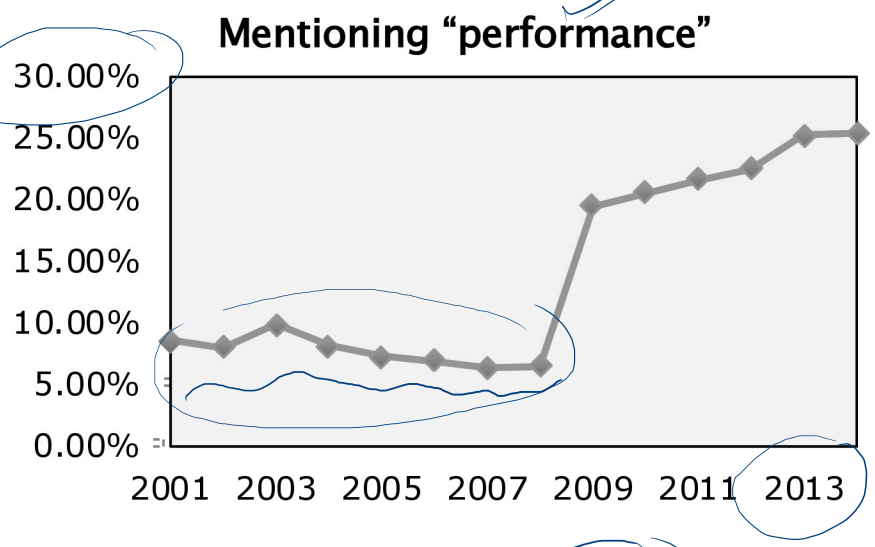
FLOPS --> Number of FLOP/Second

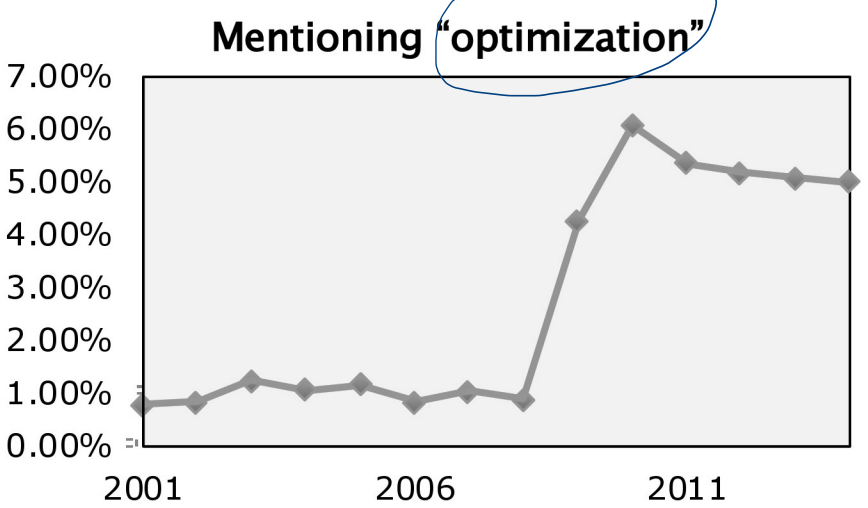
FLOPS (flops per second) can be used to characterize a computing system (peak theoretical performance) and it can also be used to characterize a program (code) i.e. actual performance of the program in terms of how many flops per seconds

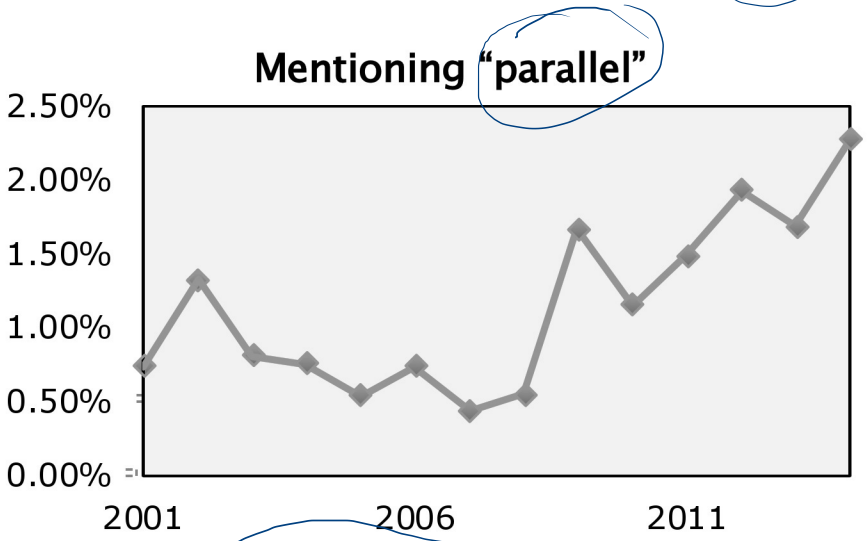


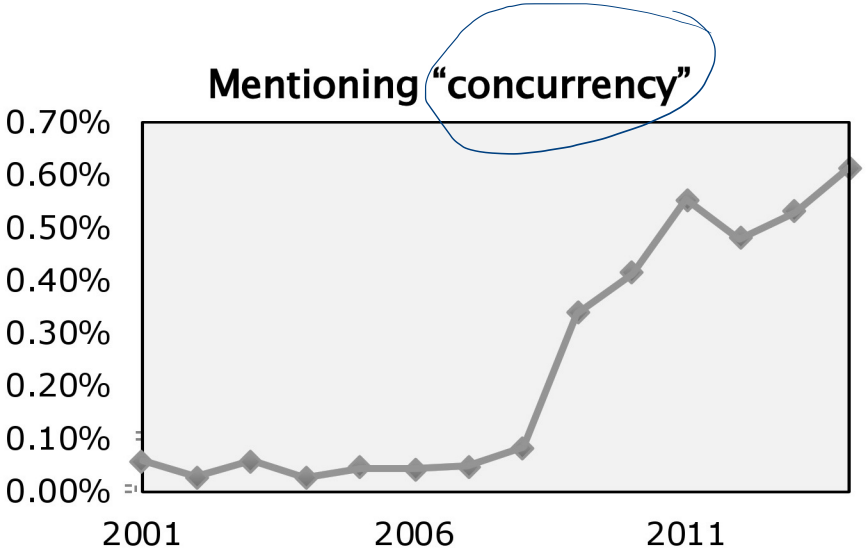


Software Developer Jobs









Source: Monster.com

High performance computing trend

1 exaFLOPS (EFLOPS) Before 2020.

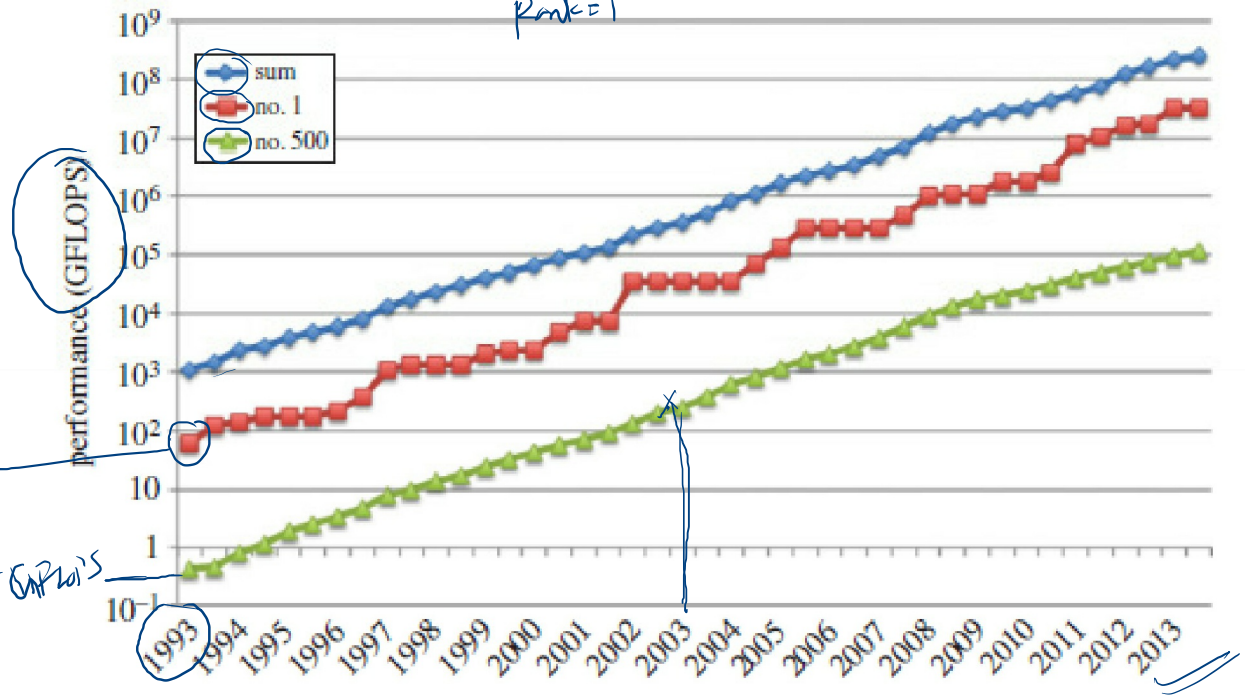


Figure 1. Data showing performance of Top500 supercomputers on the LINPACK test over the past 20 years. The top line is the sum of the top 500 systems, the middle line is no. 1, and the bottom line is no. 500 [1]. (Online version in colour.)

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

HPC trends

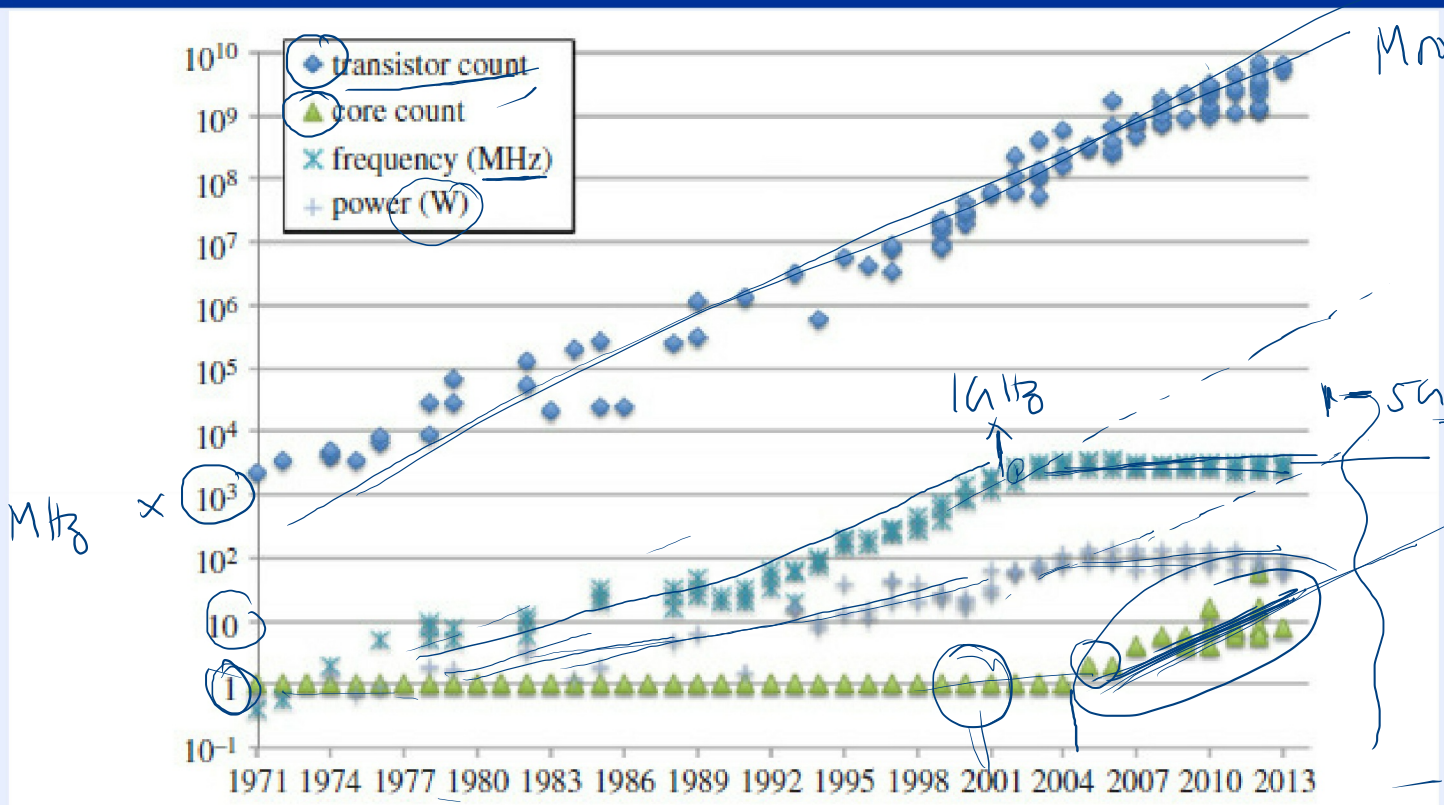


Figure 3. Evolution in transistor count, clock frequency, number of cores and power consumption for processors over the past 40 years. (Online version in colour.)

- **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

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

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 !

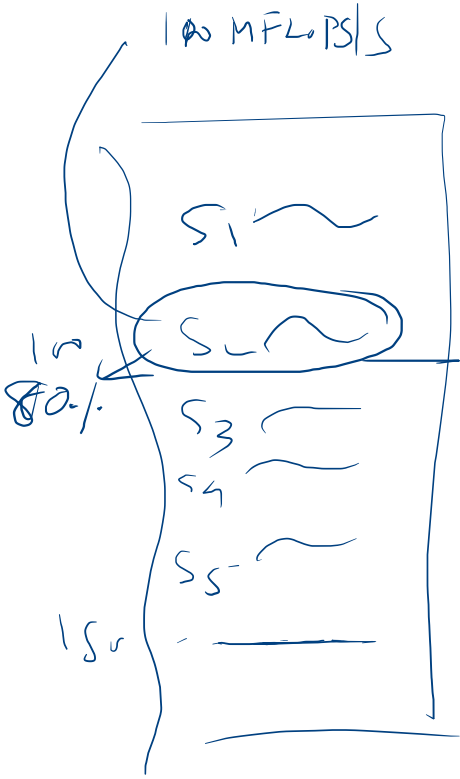
High Time to learn Parallel Programming →

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**

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”

% 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)

Solved code

% 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)

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 - 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]
```

Amdahl's Law

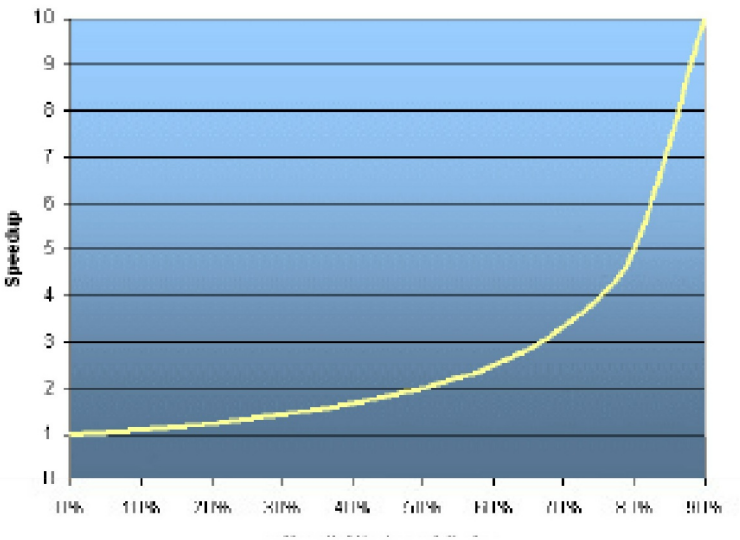
- Amdahl's Law states that potential program speedup → the fraction of code (P) that can be parallelized:

Speedup=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 %)

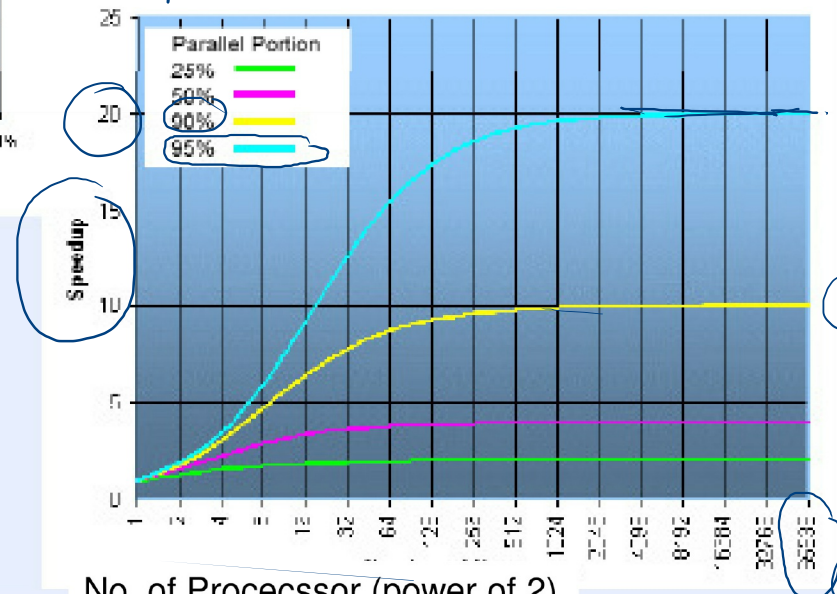
Speedup=1/(1-p)

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]

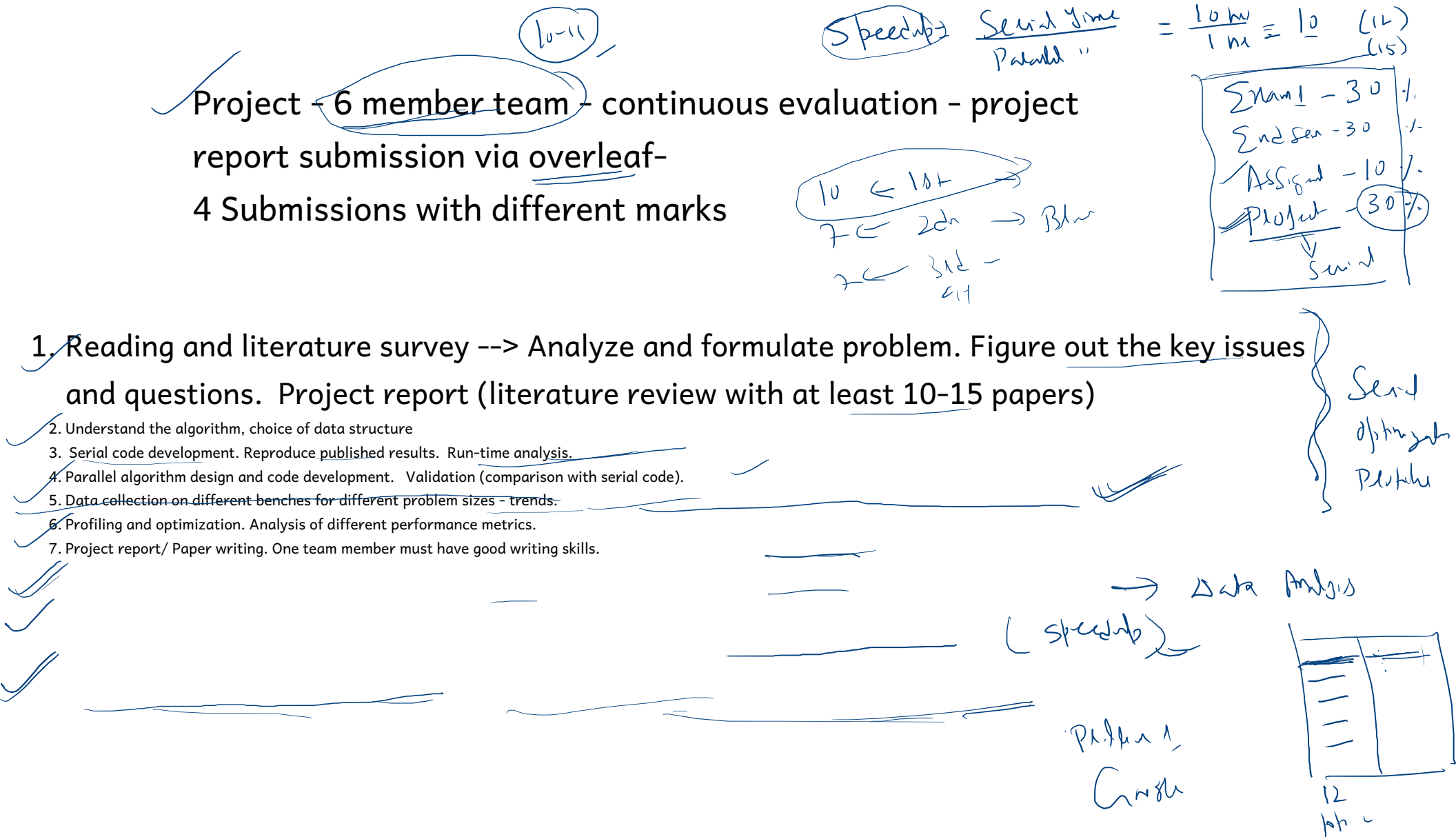


No. of Procecssor (power of 2)

Project - 6 member team - continuous evaluation - project report submission via overleaf-

4 Submissions with different marks

1. Reading and literature survey --> Analyze and formulate problem. Figure out the key issues and questions. Project report (literature review with at least 10-15 papers)
2. Understand the algorithm, choice of data structure
3. Serial code development. Reproduce published results. Run-time analysis.
4. Parallel algorithm design and code development. Validation (comparison with serial code).
5. Data collection on different benches for different problem sizes - trends.
6. Profiling and optimization. Analysis of different performance metrics.
7. Project report/ Paper writing. One team member must have good writing skills.



Practical constraints for faster serial Machines: why parallel computing?

1. **Data Transmission speeds** - the speed of a serial computer directly depends on how fast data can move through hardware. Absolute limits -> the speed of light (30 cm/nanosecond). The transmission limit of copper wire (9 cm/ nanosecond).
2. **Limits to diminishing size** - processor technology is currently allowing an increasing number of transistors to be placed on a chip. Even with molecular or atomic-level components -> a limit will be reached on how small components can be.
3. **Commercial limitations** - very expensive to make a single processor faster. Using a **larger number of available processors** to achieve a better performance is less expensive. → **Parallel processing.**

OLD : Hardware, Architecture, Compilers → Speed
NEW : Parallel programming -> Speed

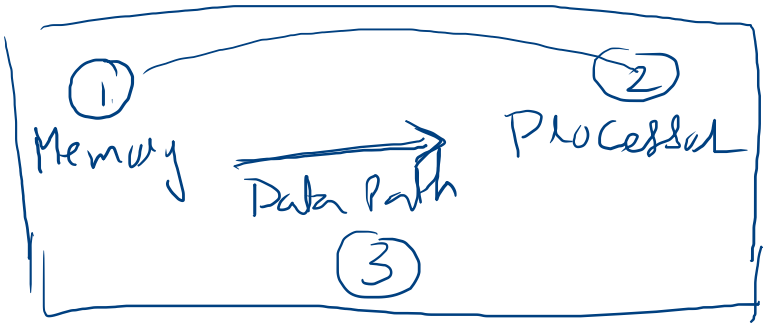
New Trends (Reinterpretation of Moore's Law):

- Number of cores per chip can double every two years
- Clock speed will not increase (possibly decrease)
- Systems with millions of concurrent threads (e.g. GPUs)
- Inter-chip parallelism (MPI) and intra-chip parallelism (OpenMP)

CUDA

Modern Processor and Principle of multiplicity

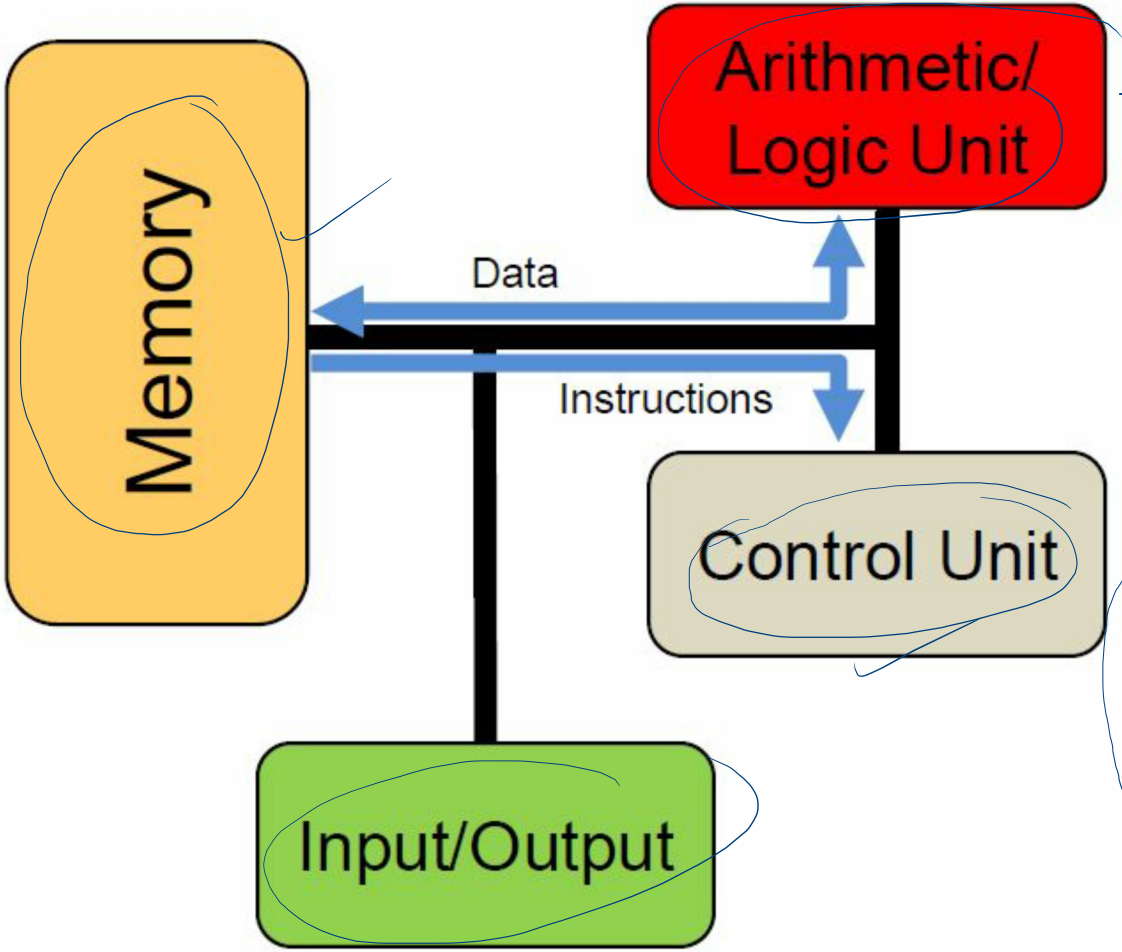
1. Multiple memories (cache, RAM, etc.)
2. Multiple Processors
3. Multiple data-paths



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



Main Performance Limitation- ?

Terminology

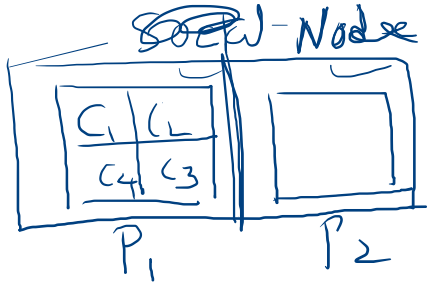
- The cores perform FLOPS.
- A processor contains one or more (CPU) cores.
- A socket holds one processor.
- A node contains one or more sockets.

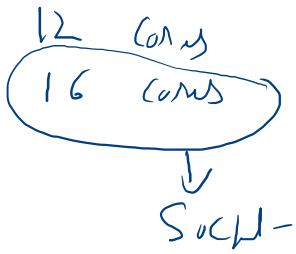
Supercomputer = several nodes → HPC

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)?



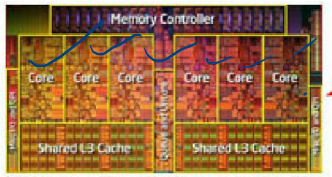


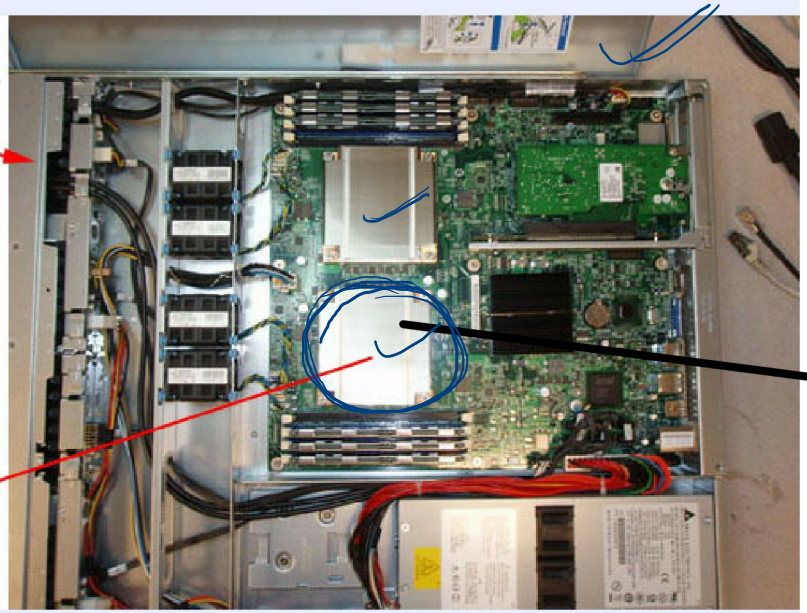
Terminology --- visual

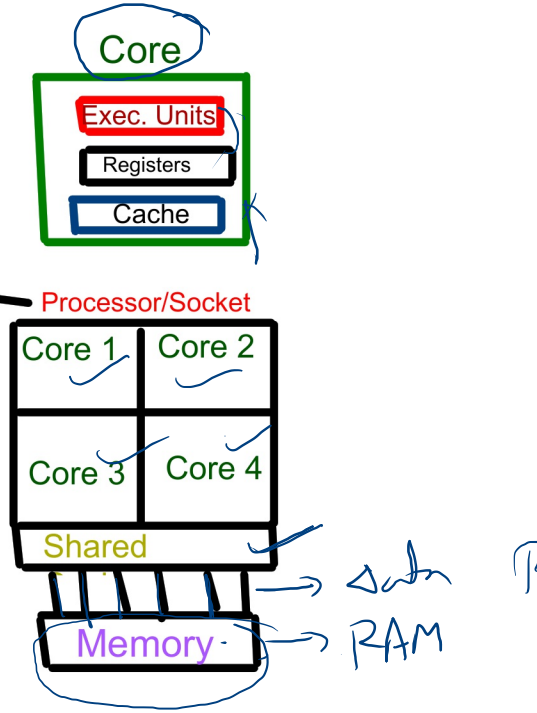
Supercomputer - each blue light is a node

Single O/S — Node - standalone Von Neumann computer

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







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

Measurement Metrics

- Flop: floating point operation (usually double precision)
- Flop/s: floating point operations per second
- Bytes: size of data (a double precision floating= 8 bytes)

- Typical sizes
- Mega Mflop/s = 10^6 flop/sec
- **Giga Gflop/s = 10^9 flop/sec**
- Tera Tflop/s = 10^12 flop/sec

Mbyte = 2^20 = ~ 10^6 bytes
Gbyte = 2^30 ~ 10^9 bytes
Tbyte = 2^40 ~ 10^12 bytes

- Peta Pflop/s = 10^15 flop/sec
- Exa Eflop/s = 10^18 flop/sec
- Zetta Zflop/s = 10^21 flop/sec
- Yotta Yflop/s = 10^24 flop/sec

Pbyte = 2^50 ~ 10^15 bytes
Ebyte = 2^60 ~ 10^18 bytes
Zbyte = 2^70 ~ 10^21 bytes
Ybyte = 2^80 ~ 10^24 bytes

Quiz:

We want a speed-up of 90x with 100 processors, the sequential part can only be _____?

Approach →

Quiz:

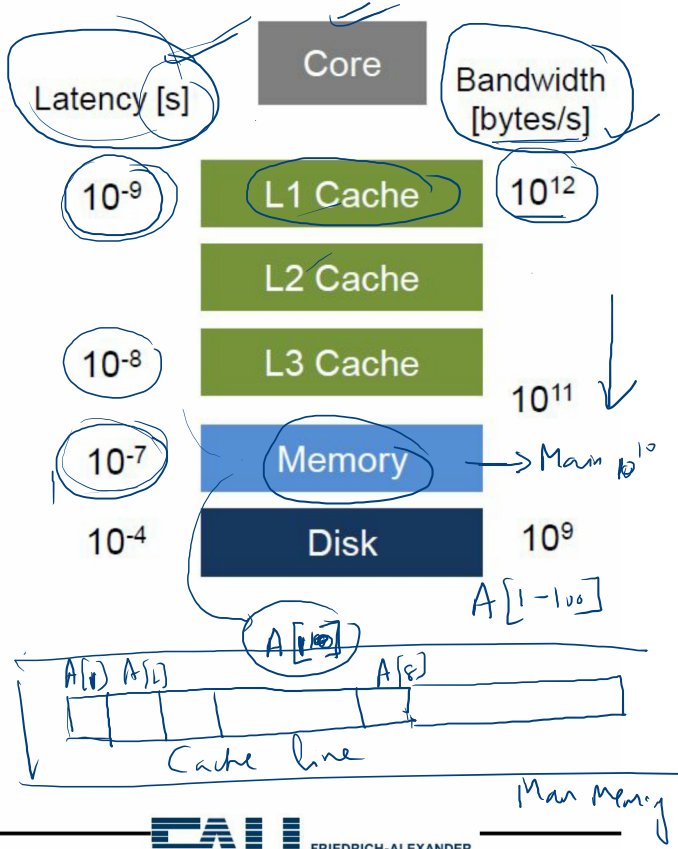
We want a speed-up of 90x with 100 processors, the sequential part can only be _____?

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

$$\rightarrow \text{Fraction time affected} = \frac{89}{89.1} = 0.999$$

Memory hierarchy

- Data transfers are the #1 limiting factor in computing
  - Main memory is too slow to keep up with the CPU's hunger for data
- You can either build a small and fast memory or a large and slow memory
  - Caches hold often-used data for fast reference
  - Multiple levels (the larger the slower)
  - Data transfers occur in "bursts" of single cache lines (typically 64 bytes)
- The purpose of many optimizations is to avoid slow data paths



Characterization of data paths

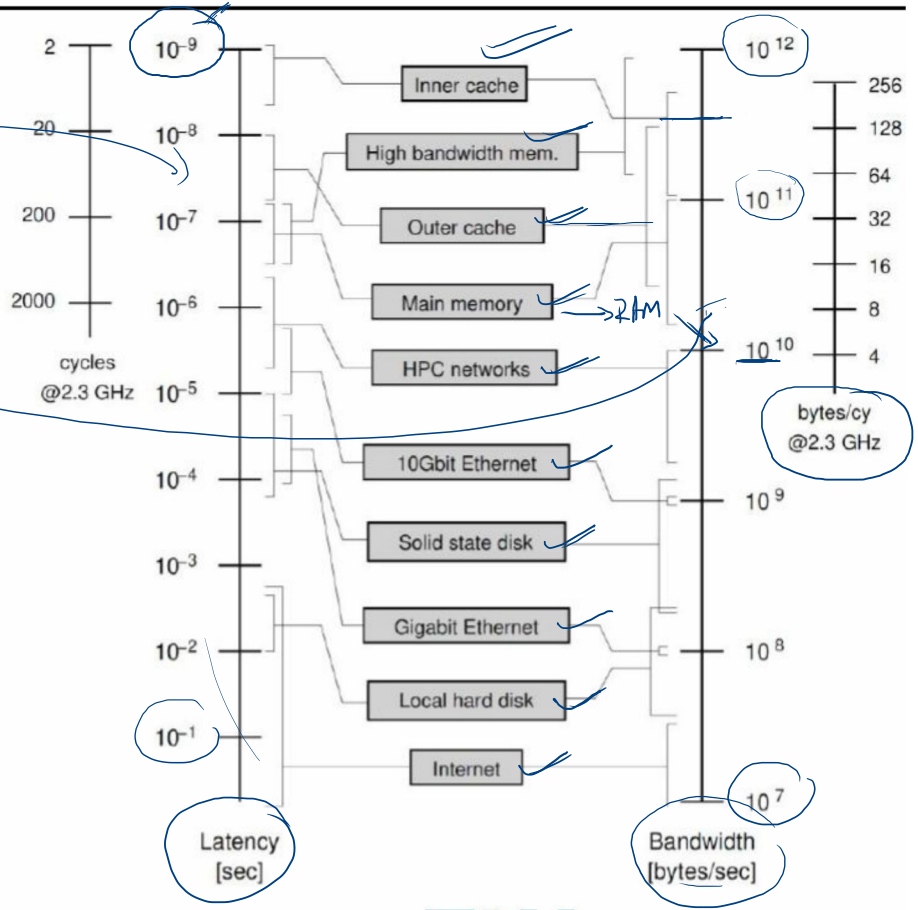
- Basic model: Latency & bandwidth
  Transfer time for message (N bytes)

$$T = \lambda + \frac{N}{b}$$

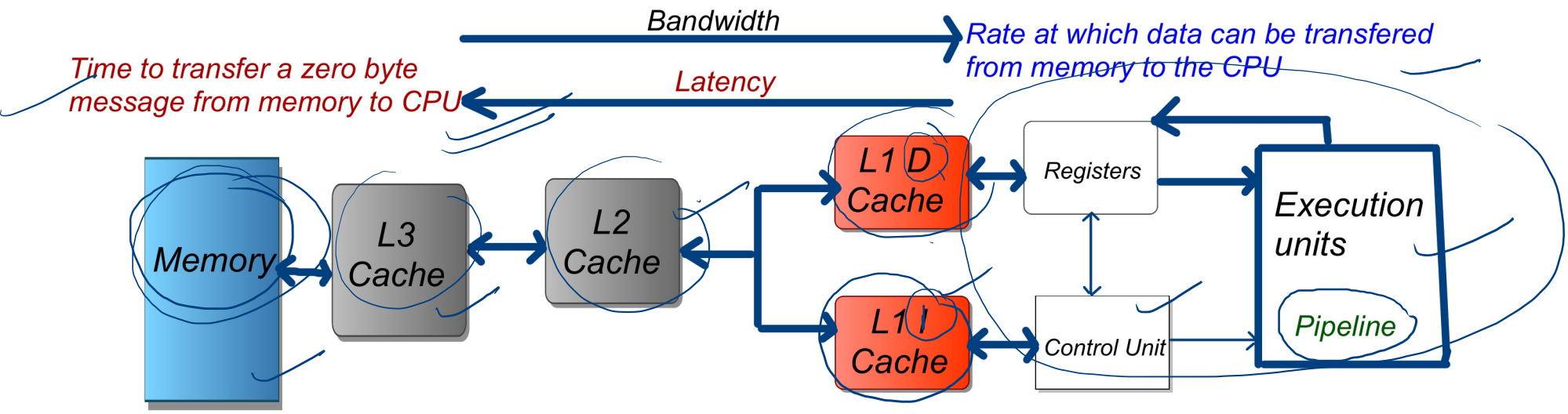
λ: latency (set-up time) [s]
b: bandwidth of data path [byte/s]

- Effective bandwidth of message transfer:

$$B_{\text{eff}} = \frac{N}{T} = \frac{N}{\lambda + \frac{N}{b}}$$



Memory Hierarchies



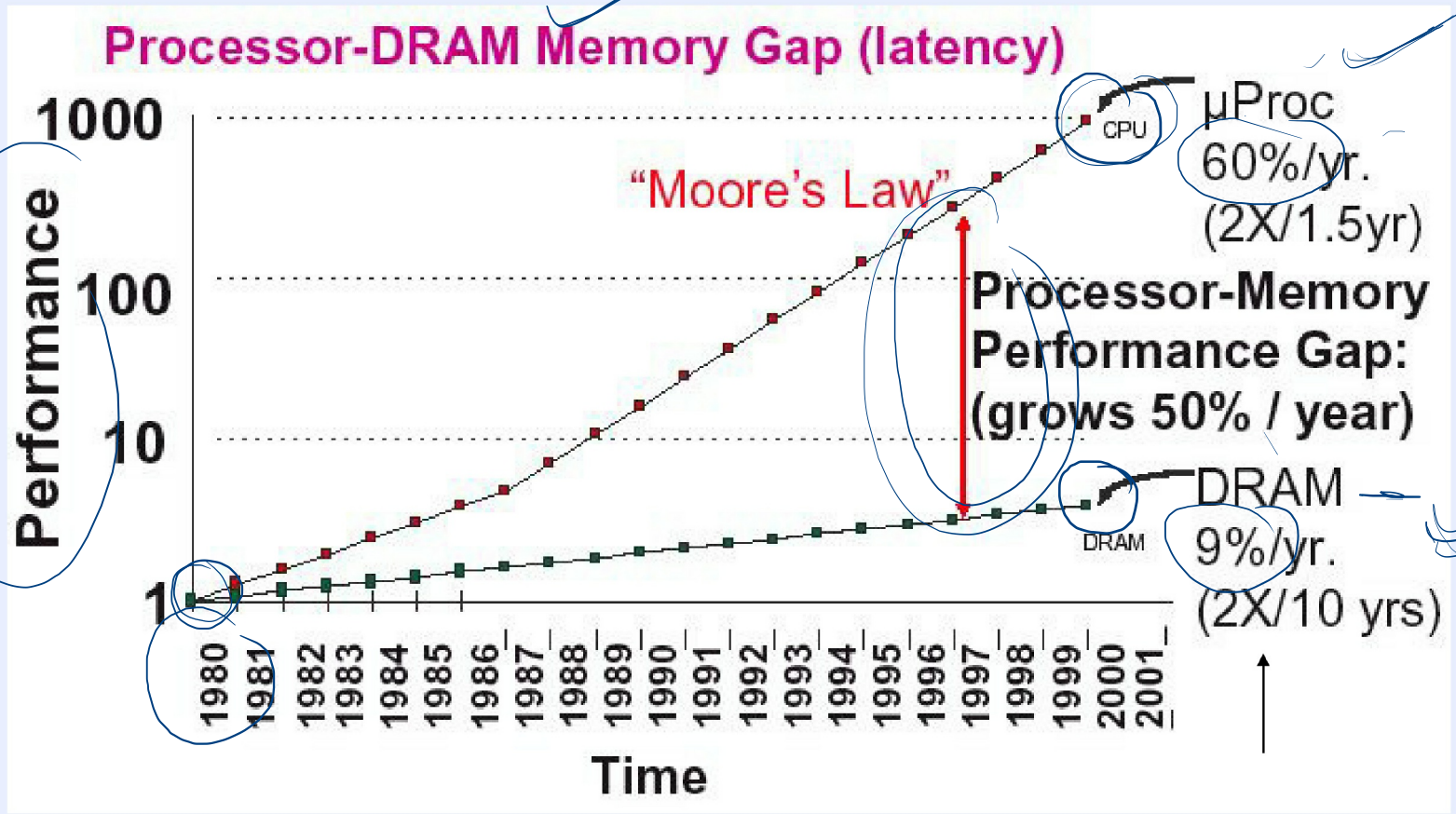
DRAM Gap - increasing distance between CPU and memory in terms of latency and bandwidth
Cache alleviate the effects of DRAM gap.

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

What's Driving Parallel Computing Architecture?



von Neumann bottleneck!! (memory wall)

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).

Machine Balance (B_m) = $\frac{\text{Possible Memory Bandwidth}}{\text{Peak Performance}}$

Hardware

» Compute

$= \frac{\text{Bytes/Sec}}{\text{Flops/Sec}} = \frac{\text{Bytes}}{\text{Flops}}$

$B_m = \frac{b_{max}}{P_{max}} \rightarrow$

$B_m P_{max} = b_{max}$

Bandwidth Based Performance Modeling

$MB = \frac{10 \text{ GB/Sec}}{10 \text{ GFLOPS/Sec}} = 1$

$= 2.5 \times 10^9 \frac{\text{cycle}}{\text{Sec}} \times 4 \frac{\text{FLOP}}{\text{cycle}} \times 1$

Lightspeed of a loop (l) = $\min(1, \frac{B_m}{B_c})$ — Machine

$B_m = B_c$

Roofline Model.

Good code

B_c ↑X ↓ SIMD

for i = 1, N

{ }

B_c

Depends on Code

if $B_m \sim B_c$: $l \simeq 1$

$\frac{B_m}{B_c} \ll 1$: $l = \frac{B_m}{B_c}$

Performance $P = l \, P_{max} = \min(P_{max}, P_{max} \times \frac{B_m}{B_c}) = \min(P_{max}, B_{max} \times \frac{B_m}{B_c})$

Machine

Code

Arithmetic Intensity: (AI, CI)

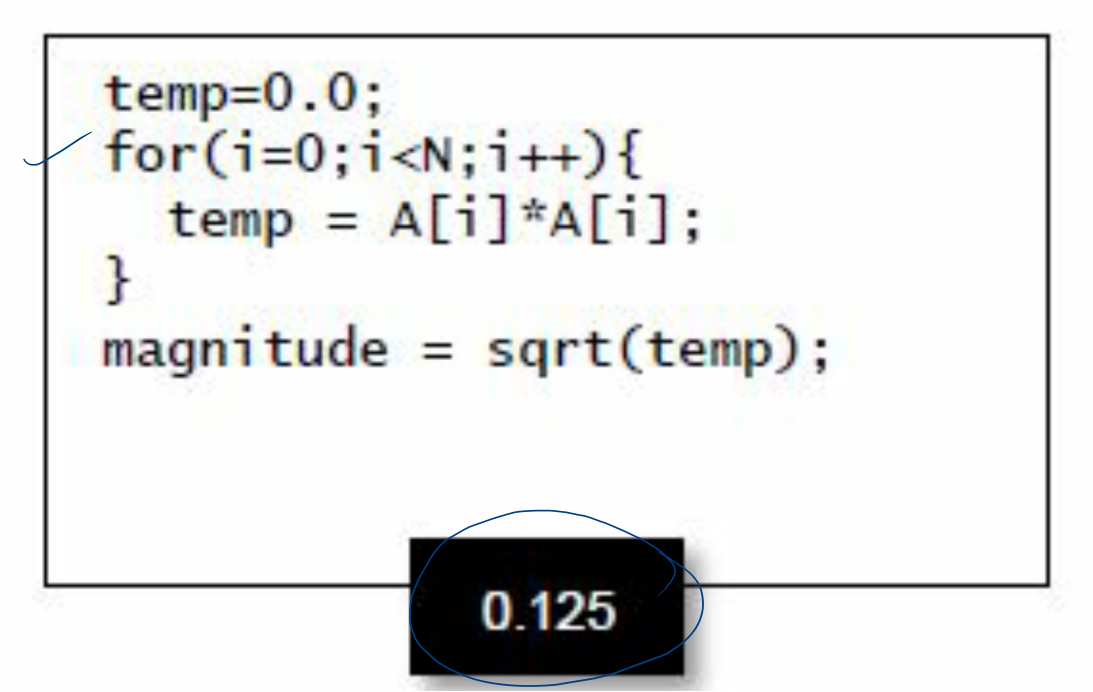
- ratio of computation to data traffic; measured in FLOPs:bytes.
- Traffic-Associated with particular memory - not the number of loads and stores.
- when arithmetic intensity exceeds machine balance (the ratio of peak floating-point performance to peak bandwidth) - likely compute bound

Lets define - Arithmetic intensity is the ratio of total floating-point operations to total **DRAM** bytes

AI=?

```
temp=0.0;
for(i=0;i<N;i++){
   temp = A[i]*A[i];
}
magnitude = sqrt(temp);
```

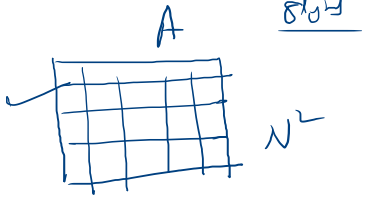
- Arithmetic intensity is the ratio of total floating-point operations to total DRAM bytes.
- Assume, N is sufficiently large that the array does not fit in cache
- The second access to A[i] exploits the cache/register locality within the processor.

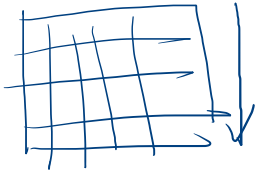


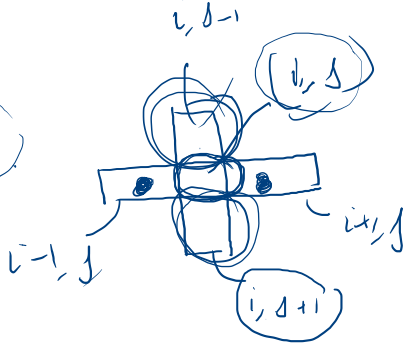
performs N flops while transferring only 8N bytes.

AI=?

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    C[i,j] = a*A[i,j] +
             b*( A[i,j-1] +
                 A[i-1,j] +
                 A[i+1,j] +
                 A[i,j+1] );
}}
```





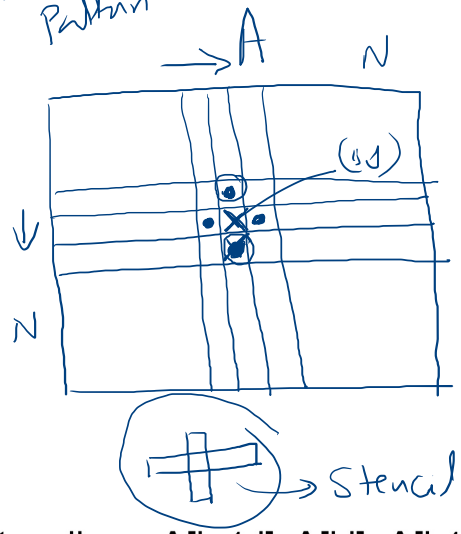


Assume cache is substantially larger than 8 N, but substantially smaller than 16 N^2

$N \gg 10$

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    C[i,j] = a*A[i,j] +
             b*( A[i,j-1] +
                 A[i-1,j] +
                 A[i+1,j] +
                 A[i,j+1] );
}}
```

0.25

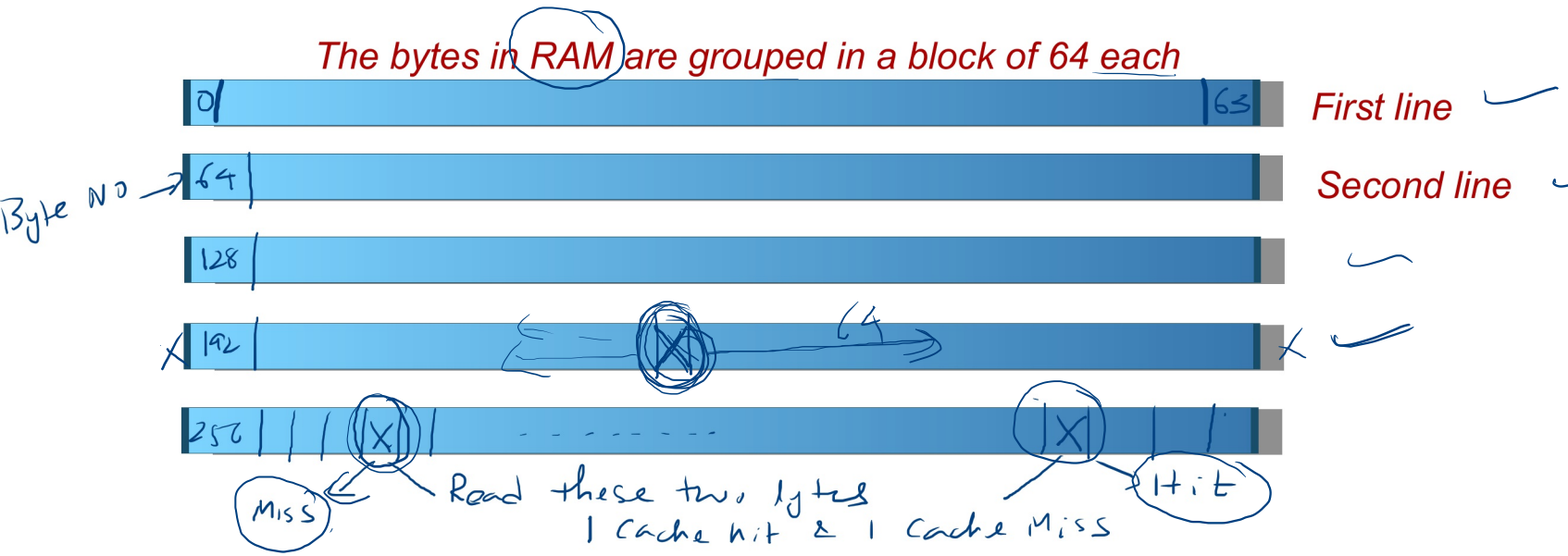


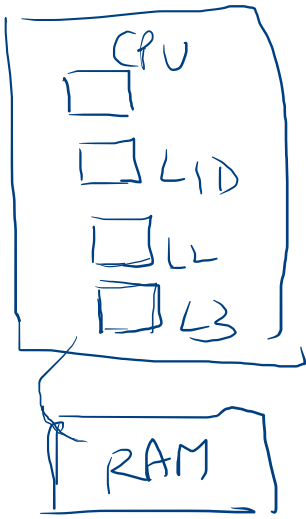
the leading point in the stencil A[i,j+1] will eventually be reused by subsequent stencils as A[i+1,j], A[i,j], A[i-1,j], and A[i,j-1].

- references to A[i,j] only generates 8 N^2 bytes of communication
- accesses to C[i,j] generate 16 N^2 bytes because write-allocate cache architectures will generate both a read for the initial fill on the write miss in addition to the eventual write back
- code performs 6 N^2 flops

Cache Lines

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





64 GB
128 GB
256 GB

Travel through the array → forward or backward →
we will get the benefit.

A[1] —— A[8]

A[ ]

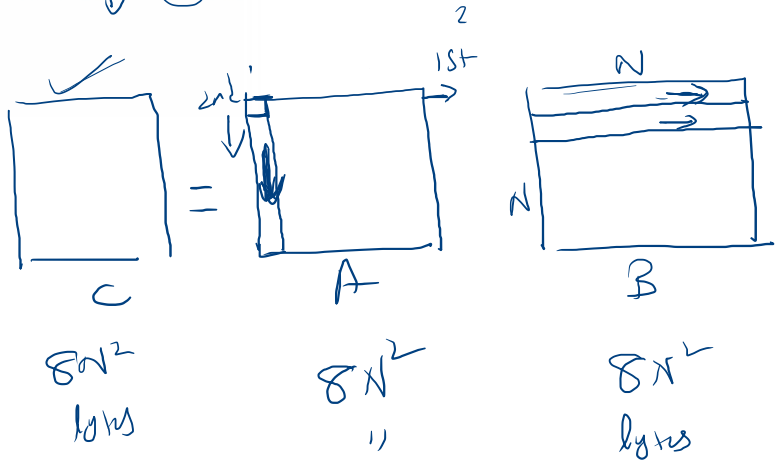
AI=?

Compute
Data Access Pattern

```
C[i,j]=0.0;
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    for(k=0;k<N;k++){
      C[i,j] += A[i,k]*B[k,j];
}}}
```

$AI = \frac{2N^3}{?}$ FLOPS

Assume the cache is substantially larger than 24 N^2,



```
C[i,j]=0.0;
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
   for(k=0;k<N;k++){
     C[i,j] += A[i,k]*B[k,j];
}}}
```

N / 16

$\frac{2N^3}{32N^2} = \frac{N}{16}$

AI ↑

A[i,j], B[i,j], and C[i,j] can be kept in cache and only their initial and write back references will generate DRAM memory traffic.

the loop nest will perform 2 N3 flops while only transferring 32 N2 bytes.

Common kernels

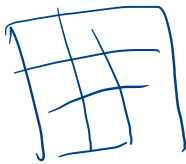
First 7 kernels/dwarfs covers most computational science applications

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

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



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

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

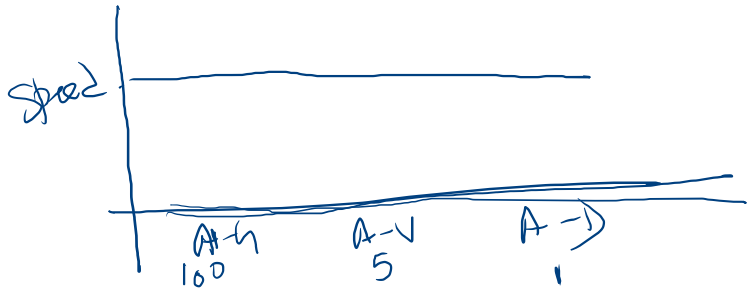
Vector Triad - popular microbenchmarking exercise
Multiply-Add nested loop on the elements of three vectors

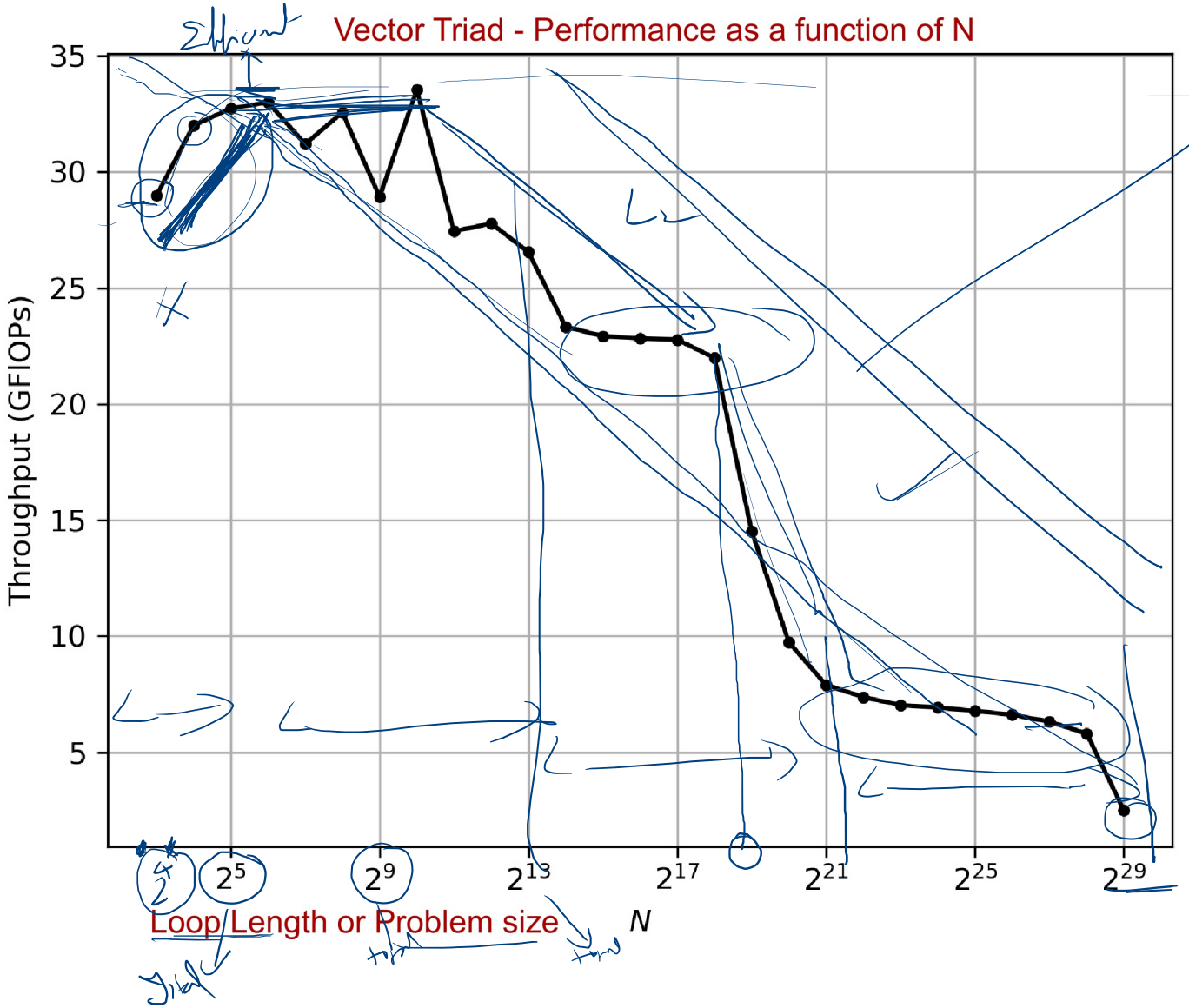
Vector Triads: A[i]=B[i]+C[i]*D[i]

```
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)/
```




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

Most important limitation – data access !!

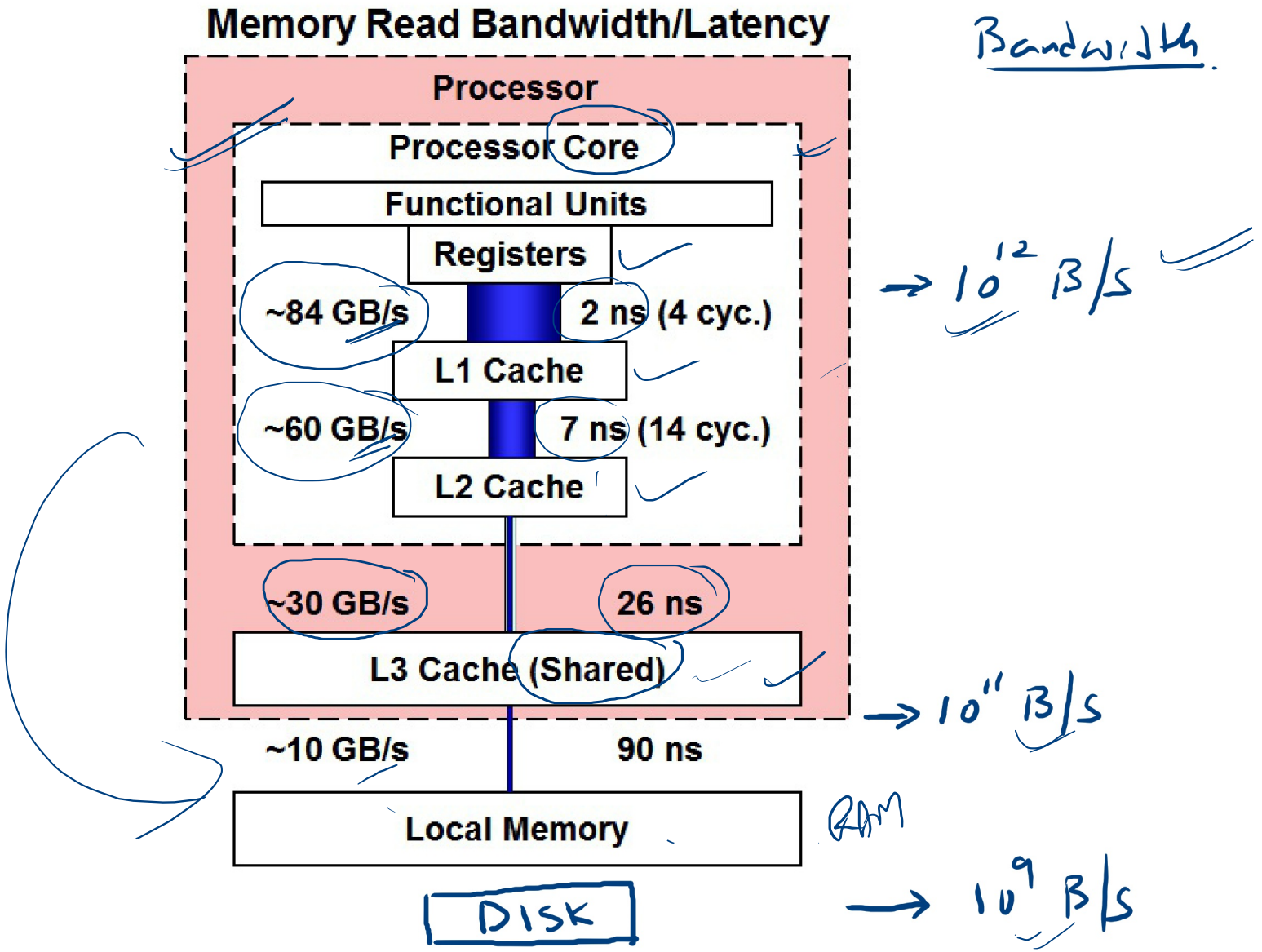
Loop based code → large amount of data movement in and out of the CPU.
Concern : underutilization of on-chip resources.

Several data paths → range of bandwidths and latencies.

Objective : reducing traffic over slow data paths.

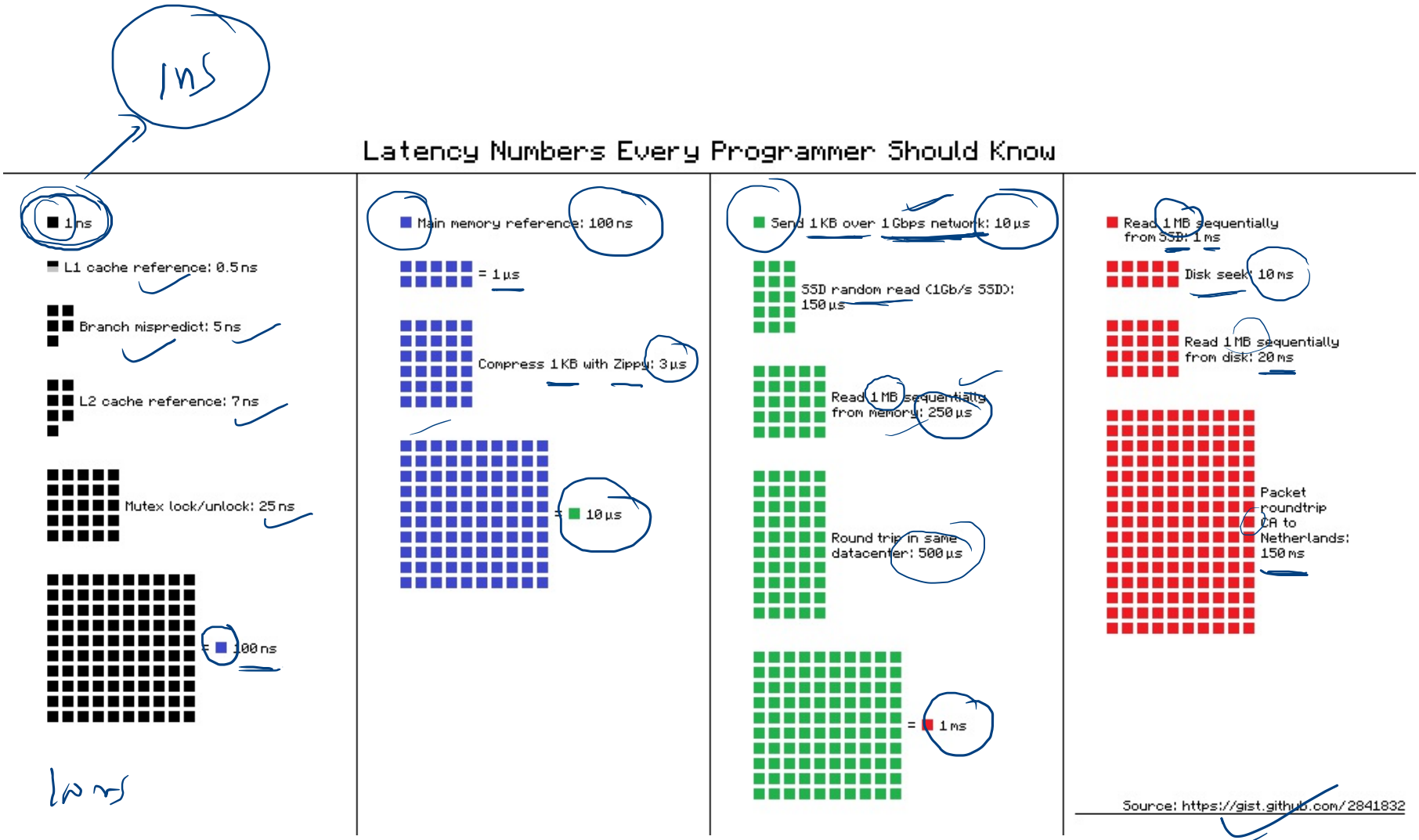
Microbenchmarking

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



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

Latency Numbers Every Programmer Should Know



Source: https://gist.github.com/2841832

Balance analysis

Theoretical performance of loop based codes.

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

Data Path	Balance (units)
cache	?
RAM	?
interconnect	?
Disk	?

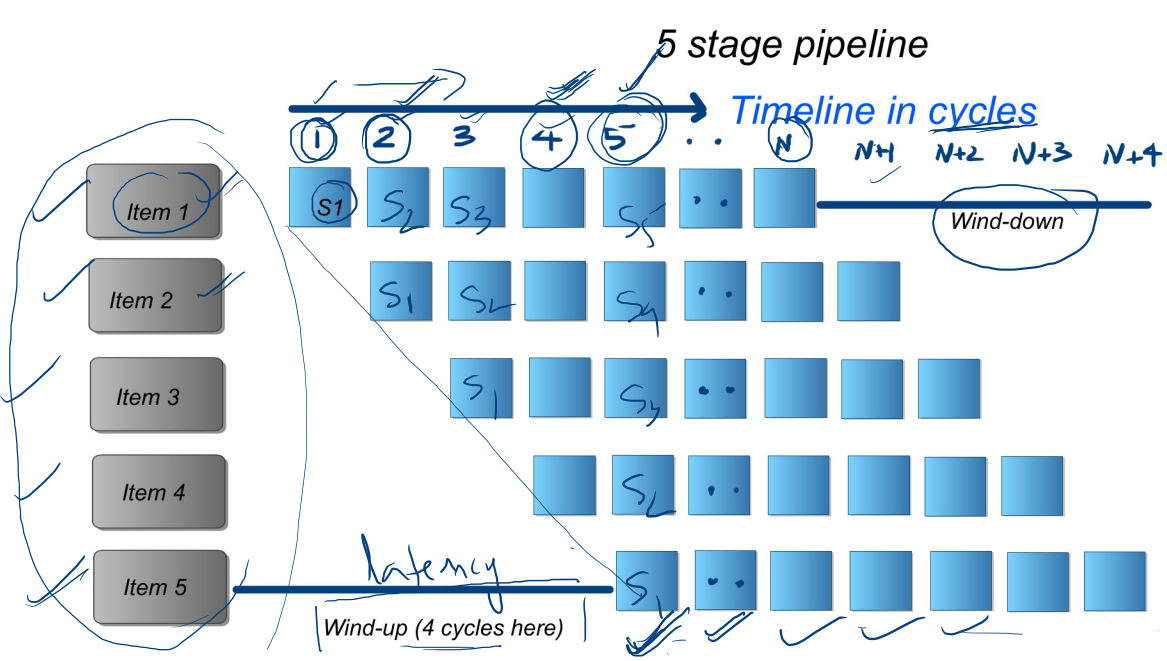
Machine balance will decrease or increase in future ?

Typical historical trend (balance) ??

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*

Vector Processor



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
Simplest pipeline --> fetch (item 1) - decode (item 2) - execute (item 3)

5 Stage pipeline —

* *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*

Dinner/Lunch

1 cycle - 10s

1 cycle = 1ms

for i = 1 N
a[i] = b[i] * c[i]
}

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

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

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

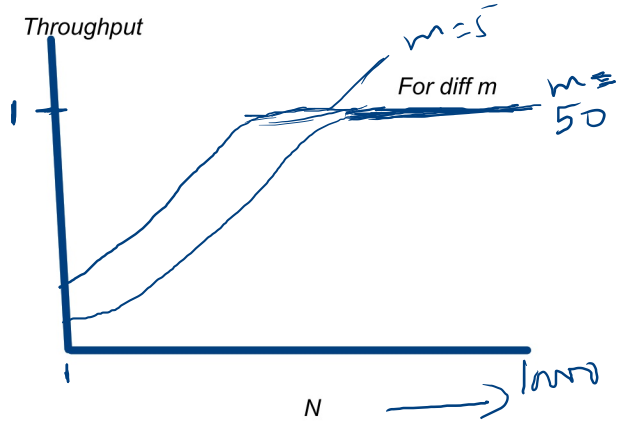
Speedup = $\frac{T_{Serial}}{T_{Pipeline}} = \frac{mN}{N+m-1} \approx m$ ($N \gg m$)

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

Compute (OPS/cycle)

$N \rightarrow$ large

Eg. Throughput $\rightarrow 1$



$N \gg m$

Addition - 5 operands

Pipeline performance is not good

p = 0.5 = $\frac{1}{1+m-1}$ N_c = 0.8

$N \gg m$

$0 \le p \le 1$

* 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

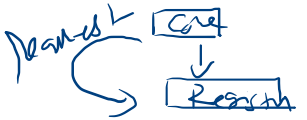
* Data transfers occur in a block (bursts) of single cache lines (~64 bytes) from one level to another level.

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)



If it does - cache hit

if it does not -> Cache Miss

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

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

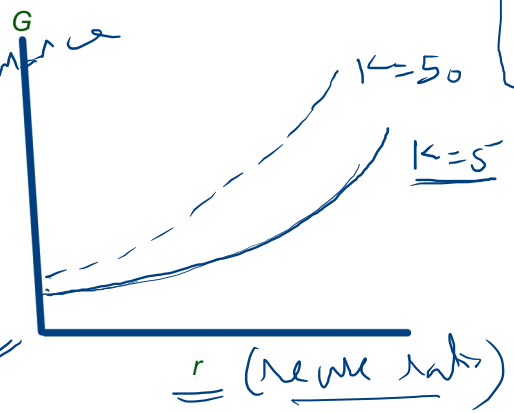
Access time to main memory - Tm

Cache access time Tc= T_m / k → $T_m = k T_c$

Average access time (for some r) $T_{av} = r T_c + (1-r) T_m$

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

$= \frac{k T_c}{r T_c + (1-r) k T_c} = \frac{k}{r + (1-r) k}$



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;
```

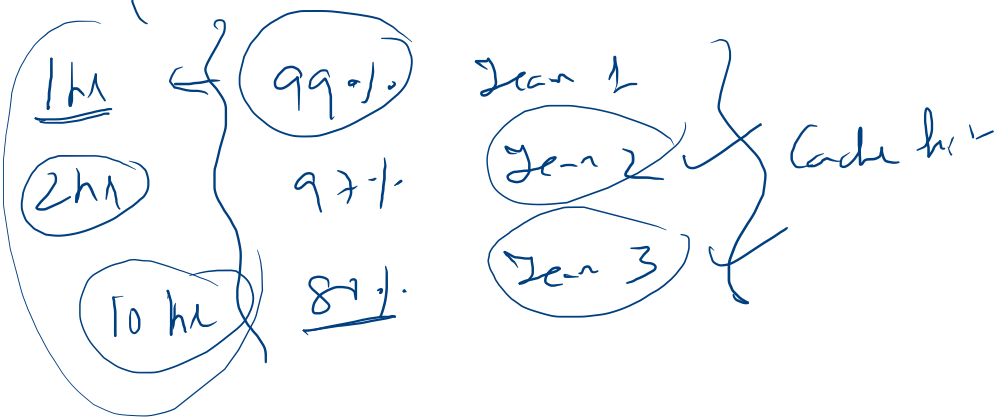
Cost of cache miss

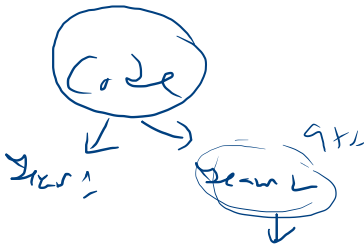
understand the difference between hit and miss
Hit time - time to deliver a line in the cache to processor (time to determine whether line is in cache included) - typical hit times - 1-2 clock cycles for L1 cache

Miss penalty - typically 10-100 of cycles

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

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





Peak Compute Performance → Clock speed × (Flops/cycle) × no. of cores ( ) → 10 GFLOPS/sec

Memory System Performance (Read throughput)

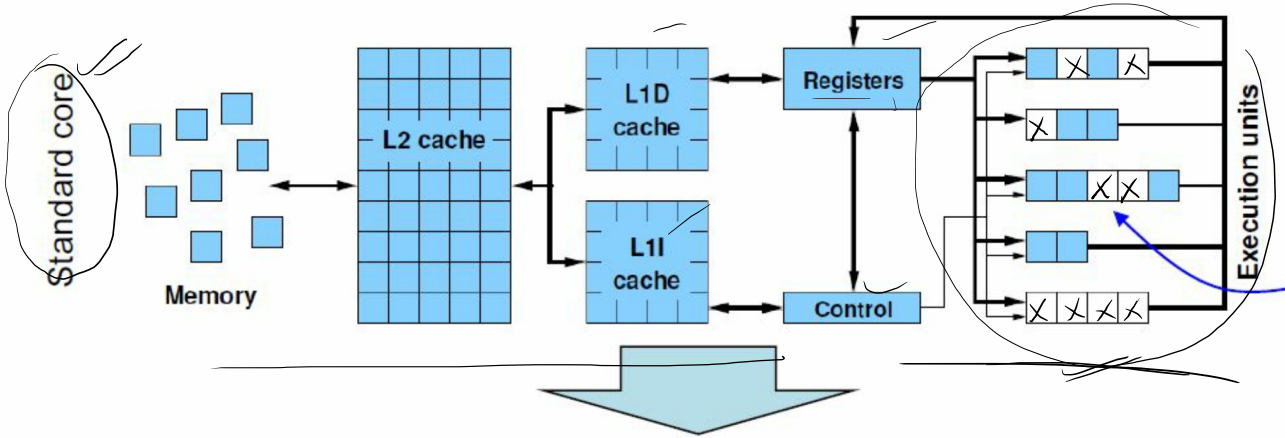
- 64 bit DDRAM interface (Double Data Rate)
- 8 channels
- 1 GHz clock freq.

Peak Access Throughput = (64 bit / transfer) × (2 transfers / clock cycle / channel) × 8 channels × (1 G clock cycles / second)

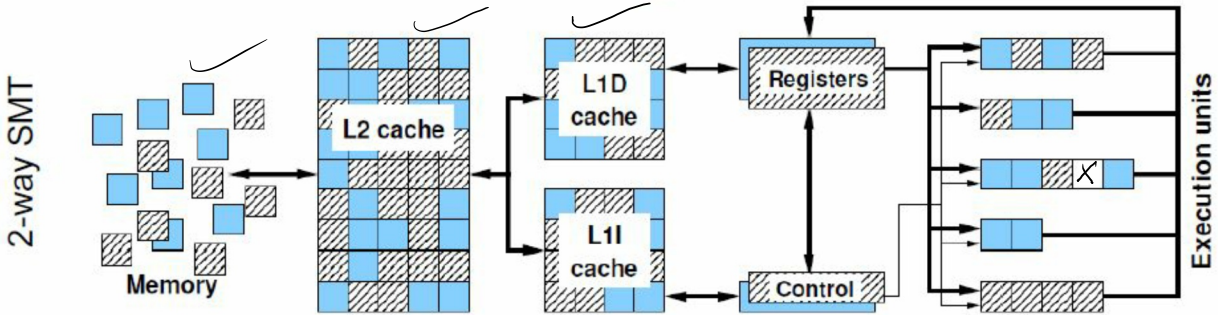
8 bytes

= 128 GB/sec

Simultaneous multi-threading (SMT)



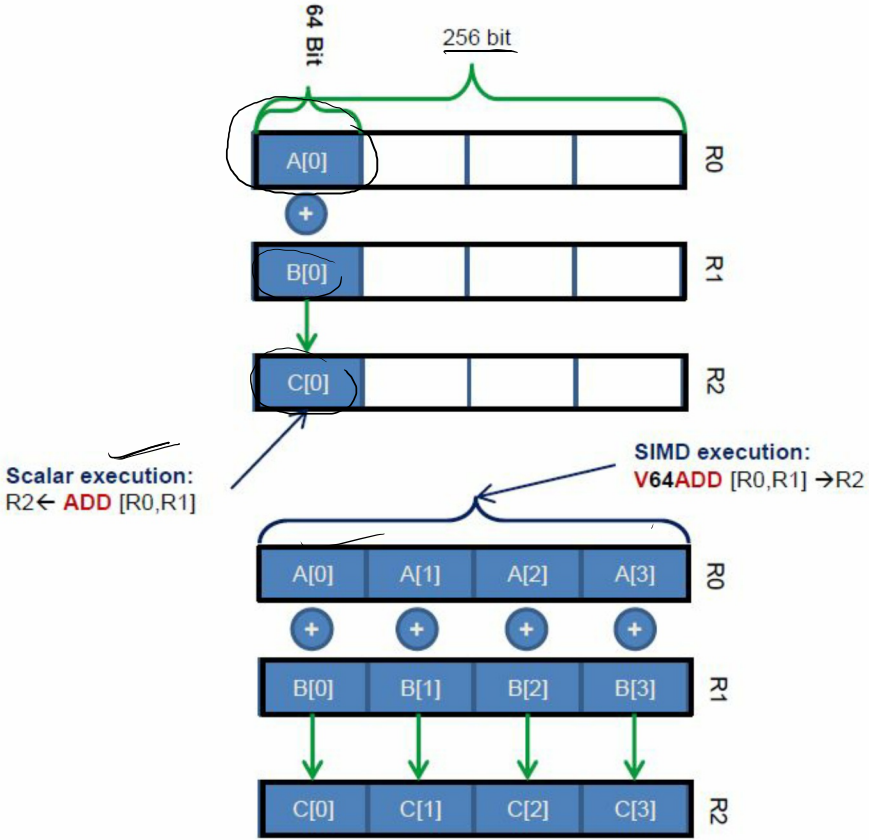
- Pipelines often underutilized due to dependencies, waiting times, etc.
- "Pipeline bubbles" mean wasted resources



- SMT can improve the utilization of pipelines
- It does not improve any other resources on the chip!
- Need to run multiple threads

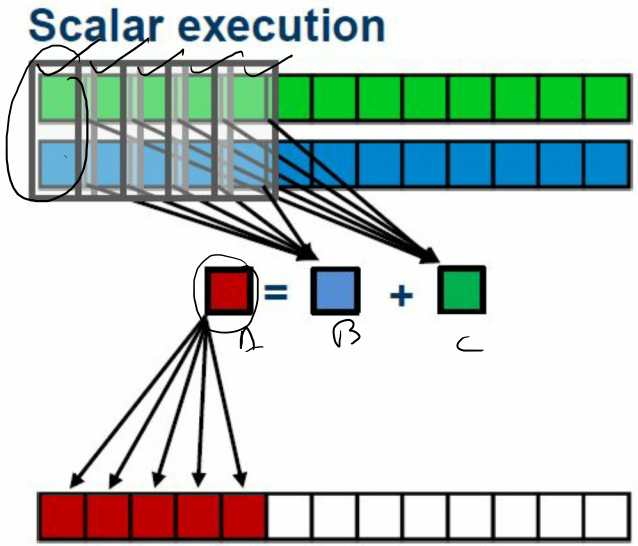
SIMD processing

- Single Instruction Multiple Data (SIMD) instructions allow the execution of the same operation on "wide" registers from a single instruction
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX(2): register width = 256 Bit → 4 double precision floating point operands
  - AVX-512: … you guessed it!
- It is not specified if these operations are concurrent
  - They mostly are, though, on modern standard CPUs



Scalar (non-SIMD) execution

```
double *A, *B, *C;
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```



Register width:

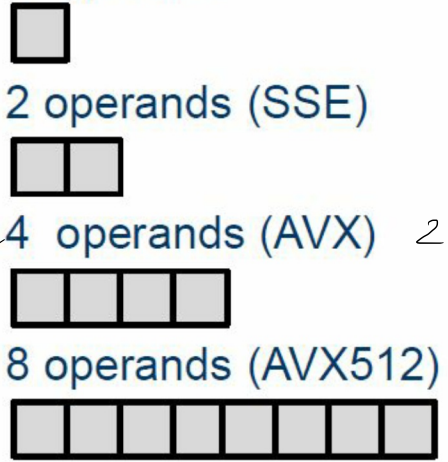
- 1 operand (scalar)

Data-parallel execution (SIMD)

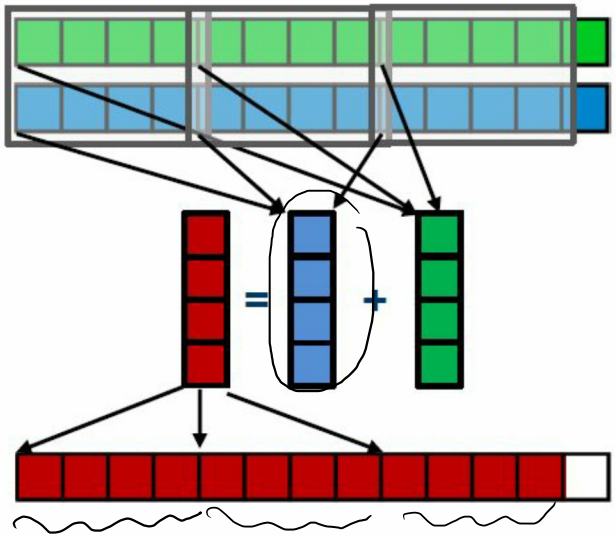
```
double *A, *B, *C;
for (int j=0; j<size; j++){
    A[j] = B[j] + C[j];
}
```

Register widths (double prec.):

- 1 operand
- 2 operands (SSE)
- 4 operands (AVX)
- 8 operands (AVX512)



SIMD execution



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
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

Dependence between
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

* 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

3 → 4

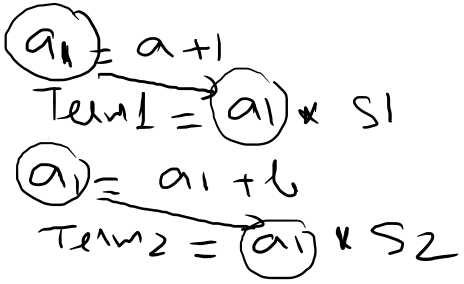
* Input dependence

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

RAW → Read after write dependence. (true dependency)



WAR → Write after Read (Anti dependency)

dependencies can be eliminated by renaming variables.

* 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]
```

```
a[0]=1
for i=1,N
{a[i] = a[i] + a[i-1]}
```

i=3
i=1

i=1 a[1] = a[1] + a[0] → T_1
i=2 a[2] = a[2] + a[1] → T_2

has to be executed in a particular order.

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

Case 1

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

dependence flows within the same iteration
dependence distance = 0

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

Case 2

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

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

a[1] = b[1] + c[1]
d[1] = a[0]
} iteration 1

a[2] = b[2] + c[2]
d[2] = a[1]
} iteration 2

Detect dependencies (rules)

1) Variable only read & never written → no dependence.

2) For each written variable → if there are accesses in other iterations than the current → dependence problem present.

* When loops has no dependence.
1) Lets say all assignments are made to arrays.
2) Each element is assigned by at most one iteration.
3) No iteration reads elements assigned by any other iteration.

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

i=1 `a[1] = a[1] + a[0]`
i=2 `a[3] = a[3] + a[2]`
i=3 `a[5] = a[5] + a[4]`

→ Can be parallelized
No dependence

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

i=0 `a[0] = a[0] + a[N/2]`
i=1 `a[1] = a[1] + a[N/2+1]`
$i = N/2 - 1$

→ No dependence.

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

i=0 `a[0] = a[0] + a[N/2]`
i=1 `a[1] = a[1] + a[N/2+1]`
$i = N/2$

→ Yes dependence
Not parallelizable

Loop carried dependences

Approach:
Look into the pattern for some iterations

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

Can this loop be parallelized?

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 writttten - 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]
}
```

Approach:
Look into the pattern for some iterations

Can this loop be parallelized?

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

Approach:
Look into the pattern for some iterations

Can this loop be parallelized?

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

Approach:
Look into the pattern for some iterations

Can this loop be parallelized?

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

Approach:

Can this loop be parallelized?

```
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

```
(1) sum =a1+1
(2) b1=sum *c1

(3) sum =a2+1
(4) b2=sum *c2
```

4 threads X

Case 2

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

i=0

```
x = (b[0] + c[0]) / 2
a[0] = a[1] + x
```

i=1

```
x = (b[1] + c[1]) / 2
a[1] = a[2] + x
```

i=3

i=2

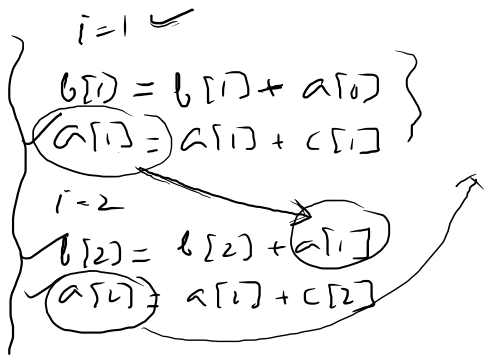
```
x = (b[2] + c[1]) / 2
a[2] = a[3] + x
```

Can this loop be parallelized?

```
for (i=0, i<N, i++) {
  a2[i] = a[i+1]
}
```

is there dependence issues.

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



Summary: Loop carried dependency.

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

```
i=1
a[1] = a[1] + c[1]
b[2] = b[2] + a[1]
i=2
a[2] = a[2] + c[2]
b[3] = b[3] + a[2]
i=3
a[3] = a[3] + c[3]
b[4] = a[3]
```

i=1 (T1)

i=2 (T2)

i=3 (T3)

for (i=0, i<N, i++)

for (j=0, j<N, j++)

a[i, j+1] = a[i, j] + 1

} Can be parallelize

i=0 → 0

i=1 → 1

i=2 → 2

Can parallelize, but not on j

Quiz

?

Code 1

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

Code-2

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

Proper Replacement

N = 10^9

a[0] = b[1]
c[1] = a[1] } i=1

a[1] = b[2]
c[2] = a[2] } i=2

a[0] = b[1]
a[1] = b[2]
a[2] = b[3]

c[1] = a[1]
c[2] = a[2]
c[3] = a[3]

Buffer Locality

Loop fusion

Loop distribution

the loop can be parallized

* Assuming both are giving same results
* Both are doing same amt of computation.

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;
```

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
```

Do less work

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
```

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)

$a = a + b^2$

↳ Code ⇒ one to one translation

Add, multiplication
DIV / SQRT / SIN / COS / TAN

} Cost difference?

for i=1, 10^8

B**2 = exp{2 ln(B)}

Lookup Table
SIN / COS / TAN

} Know the range of arguments

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
```

```
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
```

Case A

Locality is better.

Case B

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

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

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*

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.

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!

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

The Landscape of Parallel Computing Research: A View from Berkeley[8]

- 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