Parallel and Distributed Computing

Scalibility

↳ Property of system to handle growing amount of work by adding resources to the system

↳ adding resources to a system to handle much more work

e.g// 1000 customers 2 staff

10,000 customers, so need to increase staff

SCALIBILITY DIMENTIONS

Physical/Load Scalibility

↳ a system scalable w.r.t its size

↳ easily add/remove users and resources

↳ if work load decreases remove resources

Administrative scalibility

↳ ability for an increasing no of organisations/users to a system

↳ if made scalable then has the administration/organisation increased along with it or not

Functional scalibility

↳ ability to enhance the system by adding new functions w/o disrupting existing activities

Geographic scalibility

↳ ability to maintain effectiveness during expansion from local area to larger region

↳ distributed system

Generational scalibility

↳ ability of a system to scale by adopting new generations of components

↳ if system of 8 gen can we add a component of a newer gen?

→ scale out

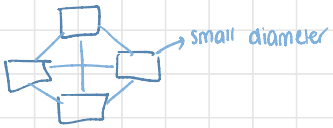
Horizontal scalibility

→ scale up

Vertical scalability

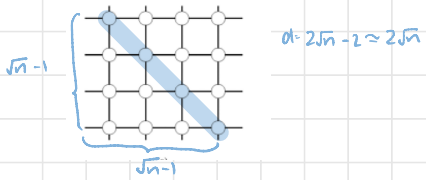
NOT completed

→ Complete Network

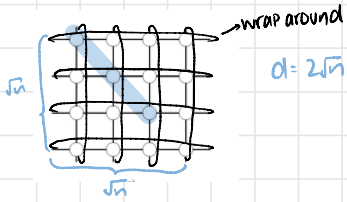


→ Straight Path

→ 2D Mesh



→ 2D Torus



Parallel computing systems

↳ doing things simultaneously

↳ task is split into parts

↳ each part is further broken into stream of instruction

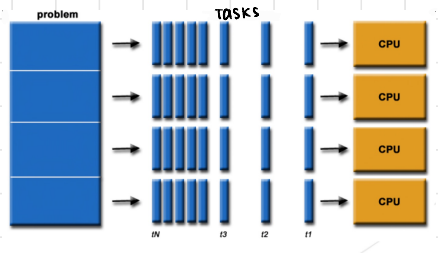
↳ instructions of different parts are then executed concurrently

PROS

↳ save time

↳ solve large problems

↳ provide concurrency



USES

↳ processors aren't getting faster

↳ solves problems of huge memory requirements

APPS

↳ data mining

↳ web search engines

↳ advanced graphics

↳ AI, ML, DL

↳ real time systems

Implicit parallelism refers to the ability of the compiler to perform certain operation in parallel without "any" hint from the programmer. This is achievable if the computation you want to perform has certain properties (e.g. no data dependencies among the parallel jobs)

Serial computing

↳ task split into stream of instructions

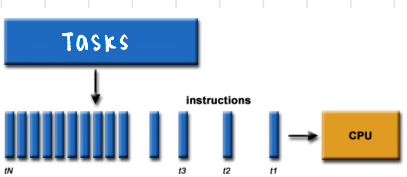
↳ they are executed seaquenhally (one after the other) on a single processor

CONS

↳ one instruction executes at a time

↳ slow

↳ expensive to make a single processor faster



PARALLEL ARCHITECHTURE

Shared memory

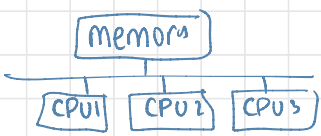
↳ large memory unit

↳ multiple processors acessing it

↳ tasks share a common address space

CON

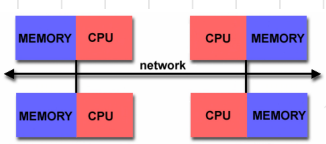
↳ not scalable → limit to no of CPUs that can be added



Distributed system

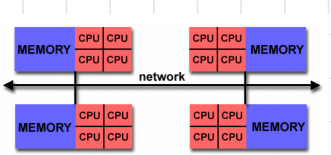
↳ multiple processors with their own memory

↳ can be scalable



HYBRID → typically used

↳ shared + distributed memory



↳ cache coherent → if 1 processors updates location in shared memory, all other processors know

↳ largest and fastest computers

PROGRAMMING MODELS

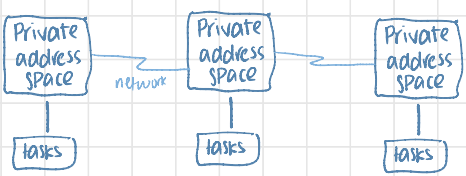
Shared memory → OPEN MP

↳ Public

Message passing → MPI

↳ private

↳ exchange messages via send/receive



Shared memory

↳ all processors have direct acess to common physical memory

PROS

↳ user friendly

↳ fast and uniform data sharing

CONS

↳ non scalable memory

↳ expensive

Distributed memory

↳ network based memory acess to NON common physical memory

↳ each processor has its own local memory

↳ no cache coherency

PROS

↳ scalable memory

↳ fast memory acess

↳ cost effective

CONS

↳ difficult to map some ds

↳ NUMA

UMA

(Uniform Memory Acess)

↳ identical processors

↳ equal acess time to memory

↳ cache coherent (if 1 processors updates location in shared memory, all other processors know)

NUMA

(NON Uniform Memory Acess)

↳ 1 SMP can access another SMP (symmetric multi processor)

↳ not all processors have equall acess time

↳ slow memory acess

Parallel overhead

↳ amount of time to coordinate parallel tasks as opposed to doing useful work

↳ include factors

- ↳ Task start up time
- ↳ Synchronizations
- ↳ Data communications
- ↳ Task termination time

Massively parallel

↳ hardware that has many processors

Scalibility

↳ property of system to handle growing amount of work by adding resources to the system

↳ adding resources to a system to handle much more work

e.g. 1000 customers 2 staff

10,000 customers, so need to increase staff

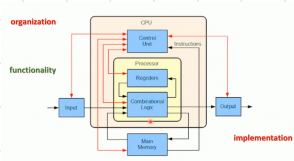
↳ factors contribute to scalibility

- ↳ hardware
- ↳ Application algorithm
- ↳ Parallel overhead

Architecture

↳ set of rules that describe functionality, organisation

↳ balancing performance ↳ efficiency ↳ cost ↳ reliability



categorisation of things

FLYNN'S TAXONOMY

↳ 4 classifications based on no of concurrent instructions and data streams available

SISD → single instruction single data

↳ no parallelism

↳ single control unit

↳ one operation at a time

PROS

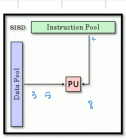
↳ no issue of complex comm b/w no of cores

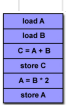
↳ requiress less power

CONS

↳ speed is limited

↳ not suitable for larger applications





eg
↳ single CPU workstations

SIMD → single instruction multiple data

↳ instructions can be executed sequentially (pipelining) or parallely (multiple functional units)...

PROS

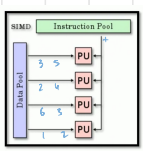
↳ increasing no of cores increases throughput (no of executions in a single time)

↳ processing speed > SISD

CONS

↳ complex comm b/w

↳ cost > SISD



USES

↳ image processing

↓

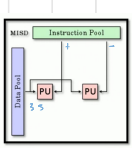
vector processors

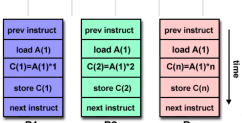
MISD multiple instruction single data

↳ performs different instructions on same data set

USES

↳ multiple cryptography algos to crack code





↳ pipelined computers

common tasks!

MIMD → multiple instruction multiple data

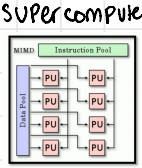
(synchronous/asynchronous deterministic/non deterministic)

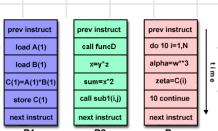
↳ performs different instruction on different data

USES

↳ most modern computers

↳ supercomputers





Parallel execution

↳ executing multiple statements at the same time

Serial Execution

↳ executing one statement at a time

↳ sequentially

Task

↳ set of instructions executed by a processor

Data parallel

↳ same operation on diff data

↳ scalable

Task parallel

↳ diff operations on same/diff data

↳ not scalable

communication

↳ data exchange for parallel tasks through

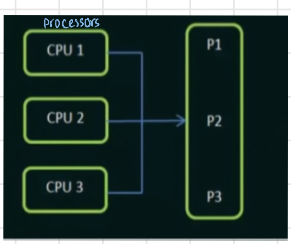
↳ shared memory bus

↳ network

Synchronization

↳ coordination of parallel tasks in real time

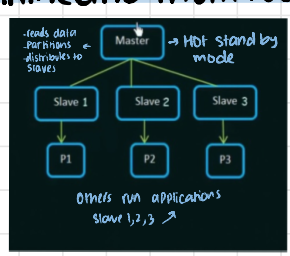
Symmetric multiprocessing



↳ shared memory

↳ all cpus similar

Asymmetric multiprocessing



↳ NO shared memory

↳ independent memory

↳ master and slave

ishma hafeez notes

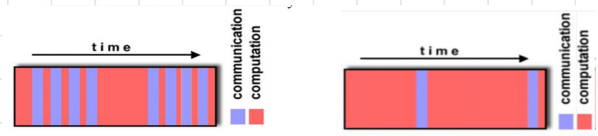
repost

Granularity

↳ ratio of computation to communication

1. Fine grain Parallelism
   - ↳ relatively small amount of computational work
   - ↳ facilitates load balancing
   - ↳ high communication overhead
   - ↳ less performance enhancement

2. Coarse grain Parallelism
   - ↳ relatively large amount of computational work
   - ↳ hard to facilitate load balancing



Amdahl's law

↳ used in parallel computing

↳ to predict theoretical speed up when using multiple processors

S latency

$$S = \frac{1}{(1-P) + \frac{P}{S}}$$

P → Parallelised Process

S → no. of Processors

- For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized.

can be Parallelized

19 hors

$P = \frac{19}{20} = 0.95$

cant be Parallelized

1 hour → critical

$\frac{1}{20} = 0.05$

min execution > that critical 1 hour

For example, suppose that we use our strategy to search for primes using 4 processors, and that 90% of the running time is spent checking 2k-digit random numbers for primality (after an initial 10% of the running time computing a list of k-digit primes). Then P = .90 and S = 4 (for 4-fold speedup). According to Amdahl's Law,

$$overall\ speedup = \frac{1}{(1-0.90) + \frac{0.90}{4}} = \frac{0.10}{0.225} = 3.077$$

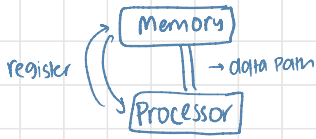
This estimates that we will obtain about 3-fold speedup by using 4-fold parallelism.

CHP 2

Pipelining

↳ each instruction is broken down into stages

↳ Process executes 1 stage in each cycle



CONS

↳ if stage slow, speed slow

↳ if too many conditional jumps, accurate branch prediction needed

↳ misprediction causes, large no of instructions to be flushed

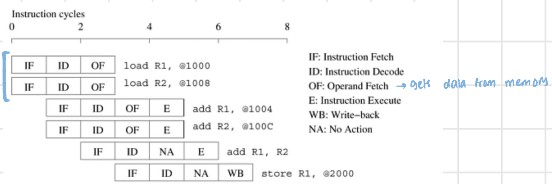
Super Scalar Execution (super Pipelining)

uses → linear algebric operations, dot product of 2 vectors

↳ Processor checks which instructions can be run together → done by processor at runtime

↳ if instructions have nothing to do with eachother it can run them parallely

↳ needs multiple logical units → as IF/ID/OF... multiple instructions at a time



CONS

↳ data dependency

↳ branching ?? 

↳ memory latency

↳ expensive

↳ complex

Very Long Instruction Words (VLIW)

→ solves data dependency issue → check next slide

↳ instructions that can run concurrently are packed into groups

↳ then dispatched together

→ done by compiler at compile time → during compile time group these instructions

SuperPipeline

CONS
- ↳ more complex h/w
- ↳ realtime → cant take time as small window

PRO
- ↳ view of dynamic state

VLIW

PROS
- ↳ more simple h/w → as not dynamic
- ↳ offline → can take time

CON
- ↳ no view of dynamic state → reduces accuracy of branch and memory prediction
- ↳ more difficult branch and memory prediction

How 2 way superscalar remain idle

Horizontal waste
↳ not all issue slots can be filled in a cycle

Vertical waste
↳ Processor issues no instructions in a cycle

(b) Execution schedule for code fragment (i) above.

(c) Hardware utilization trace for schedule in (b).

Data DEPENDENCY
result of one operation is output to next

limited performance

in order issue
issue instruction on the order they came in

if 2nd instruction is dependent on 1st only one instruction is issued in cycle

out of order
issue instruction in any order

1st and 3rd can be coscheduled and not 2nd as it is dependant on 1

Branch DEPENDENCY
↳ scheduling instructions across conditional branch statements can't be done

Resource dependency
↳ two instructions require same resource

Limitations of memory system

memory latency
higher → increase memory waiting time

↳ gap b/w rate at which processor works and time taken to fetch data from memory

solution is CACHE

memory Bandwith
↳ rate at which data is moved b/w memory to processor

↳ increasing blocksize increase bandwith

CACHE

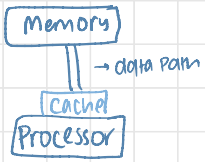
↳ smaller and faster memory

↳ sits b/w processor and main memory

WORKS

↳ if data is in cache
  get from cache

↳ if data is in main memory
  store in cache



PROS

↳ cache has lower latency (even lower if repeated data)

Temporal locality

↳ notion of repeated reference to a data item in a small time window

cache hit ratio

↳ % of memory acess found in cache

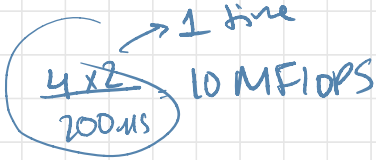
↳ it defines performance

- **Given:**
- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.
- Multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.
- **Solution :**
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200 + 16 µs.
- This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS.

only when divide

1 word = 4 byte

4 instructs per cycle = cache line = 4



storing 2 matrices into cache

= 200 µs → load

Multiplying 2 matrices → muladd

$= 2(32)^3 = 64K$ operations

$= 64K/4$ → 4 operations in 1 cycle (load, load, mul, add)

= 16K µs → muladd

load $R_1, [R_2]$ (a_{ik}) → 100 ns

load $R_3, [R_4]$ (b_{kj}) → 100 ns

mul,add R_5, R_1, R_3

↓

2 operations

no of operations = $2n^3$

no of iterations = n^3

Total time

$$\text{FLOPS} = \frac{64K}{200\mu s + 16\mu s} = 303 \text{ MFLOPS}$$

LATENCY

FLOP = no of FLOPS / time

matrix multiply

```
C=0
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    for(k=0; k<n; k++)
      c_ij += a_ik * b_kj
```

C = A × B

64 × 64 matrices

4 bytes

latency

Processor: 1 GHz

memory access time: 100 ns

word size: 4 bytes

↳ whenever one data from some location

↳ if size not given of matrice THEN USEFULL

load R_1, [R_2] (a_{ik}) → 100 ns

load R_3, [R_4] (b_{kj}) → 100 ns

= 200 ns

ignore increasing for time being

mult, add R_5, R_1, R_3

↓ 2 operations

$$FLOPS = \frac{2}{200\,\mu s} = 10 \times 10^6 \text{ FLOPS}$$

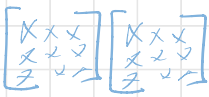
approximate

↓

10 MFLOPS

execution time

CACHE



C = A × B

64 × 64 matrices

4 bytes

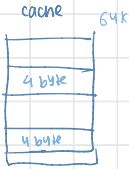
latency

MM = 100 ns

Cache = 1 ns

Cache size = 64K

word size = 4 byte



matrix A size = 64 × 64 × 4 (element in array size) = 16 K (round off) ns

B size = " 16K

C size = " 16K

Total data size = 16K + 16K + 16K

= 48 K → can all fit in cache

load all data into cache

= 100 ns × 64 × 64 × 2 → no need to load matrix c

= 800 μs

load R_1, [R_2] (a_{ik}) → 1 ns

load R_3, [R_4] (b_{kj}) → 1 ns

assume 20 ns for 1 iteration

mult, add R_5, R_1, R_3

↓ 2 operations

total no of iterations = $n^3 = 64^3$

total time = $64^3 \times 20\,ns + 800\,\mu s$

$$FLOPS = \frac{2 \times 64^3}{64^3 \times 20\,ns + 800\,\mu s} = \simeq 90 \text{ MFLOPS}$$

Storing 2 matrices into cache

= 200 μs → load

Multiplying 2 matrices → muladd

= $2(64)^3$ = 524288 ≃ 524K

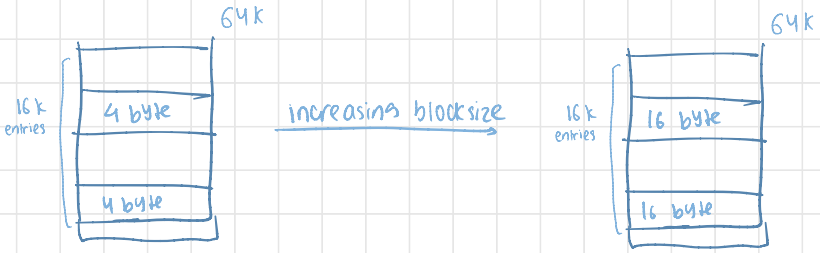
= 524K/4 → 4 operations in 1 cycle

= 131K μs → muladd

Total time

$$\text{FLOPS} = \frac{2(64)^3}{200\,\mu s + 131K\,\mu s} = 4\,\text{GFLOPS}$$

BANDWITH



DOt Product → dosent use cache

latency
MM = 100ns
word size: 4 byte

NO CACHE

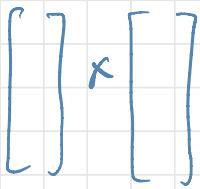
load R_1, $[R_2]^{a_{ik}}$ → 100 ns
load R_3, $[R_4]^{b_{kj}}$ → 100 ns
mulh,add R_5, R_1, R_3
↓
2 operations

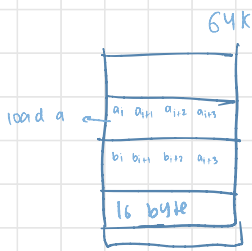
FLOPS: $\frac{2}{200ns}$ = 10 MFLOPS

latency
MM = 100ns
word size: 16 byte

CACHE

load R_1, $[R_2]^{a_{ik}}$ → 100 ns
load R_3, $[R_4]^{b_{kj}}$ → 100ns → 200ns
mulh,add R_5, R_1, R_3
↳ 2 operations
⋮
load R_1 → 1ns
load R_3 → 1s → 2ns → ignore for now
⋮
2 more times





FLOPS: $\frac{2 \times 4}{200ns}$ = 40 MFLOPS

SOLVE USING CACHE HIT RATIO

1 → MM 3 → cache
cache hit ratio = 75%
avg memory access time = $\underbrace{75}_{\text{hit ratio}} \times \underbrace{1ns}_{\text{cache time}} + \underbrace{25}_{\text{miss ratio}} \times \underbrace{100ns}_{\text{mm time}}$ = 25.75ns

FLOPS: $\frac{2}{50ns}$ = 40MFLOPS
↓
25.75 × 2
↓
2 acess to the memory

2.2 Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 1 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors? Note: Where necessary, assume an optimal cache placement policy.

```
1       /* dot product loop */
2       for (i = 0; i < dim; i++)
3               dot_prod += a[i] * b[i];
```

DOT Product of 2 vectors

cache size = 32 kB → latency = 1 ns

MM size = 512 MB → latency = 100 ns

Processor = 1 GHz (10^9)

cache line size = 4 words

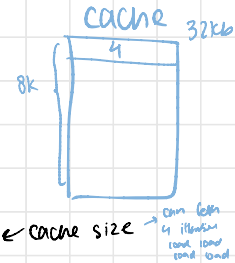
load a

load b

muladd a, b, C

↳ load from Memory

= 100 ns + 100 ns



no of miss = 2 (load a, load b)

time for miss = 2 x 100 ns = 200 ns

no of operation = 2

no of iteration = 4 ← cache size

total no of operation = 4 x 2 = 8

$$= \frac{8}{200\ ns} = 40\ MFLOPS \quad (10^{-9},\ 10^{6})$$

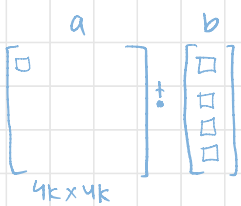
2.3 Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4K x 4K. (Each row of the mat takes 16 KB of storage.) What is the peak achievable performance of this technique usin a two-loop dot-product based matrix-vector product?

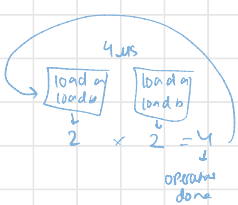
```
1       /* matrix-vector product loop */
2       for (i = 0; i < dim; i++)
3               for (j = 0; j < dim; j++)
4                       c[i] += a[i][j] * b[j];
```

Multiple matrix x vector

matrix = 4K + 4K

each row takes = 16 kB





Total storage matrix takes

= 16 kB x 4K

= 64000 kB → cant fit inside cache

Storing B in cache

= 100 ns x 4K/4

= 4,000,00 ns

= 1×10^{-4}

no of miss in a = 1

time for miss = 100 ns

no of operation = 2

no of iterations = 4

total operations = 8

$$FLOPS = \frac{8}{100\ ns} = 80\ MFLOPS$$

2.4 Extending this further, consider the problem of multiplying two dense matrices of dimension 4*K* x 4*K*. What is the peak achievable performance using a three-loop dot-product based formulation? (Assume that matrices are laid out in a row-major fashion.)

```
1         /* matrix-matrix product loop */
2         for (i = 0; i < dim; i++)
3                 for (j = 0; i < dim; j++)
4                         for (k = 0; k < dim; k++)
5                                 c[i][j] += a[i][k] * b[k][j];
```

MULTIPLY MATRIX INTO MATRIX

cache size = 32 KB → latency = 1 ns
cache line size = 4 words
MM size = 512 MB → latency = 100 ns
Processor = 1 GHz
10^9

no of miss = 5
time for miss = 500 ns
HOW

no of operation = 2
no of iterations = 4
total operations = 8

Total storage matrix takes
= 16 KB x 4K
= 64000 KB → cant fit inside cache

FLOPS = $\frac{8}{500ns}$ = 16 MFLOPS

(a) Column major data access

load a
load b
muladd a, b, c

Example 2.4 Effect of block size: dot-product of two vectors

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in Example 2.2.

MM = 100 cycles
word = 1
$\frac{1 \times 2}{200ns}$ = 10 MFLOPS

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

MM = 100 cycles
words = 4
$\frac{4 \times 2}{200ns}$ = 40 MFLOPS

Another way of quickly estimating performance bounds is to estimate the cache hit

Example 2.5 Impact of strided access

means step

A strided access pattern accesses a sequence of addresses with a uniform skip between each referenced address. For example, sequence 1, 1001, 2001, 3001, 4001, 5001, … is a strided access pattern with a stride of +1000

Strided access patterns refer to memory access patterns where data elements are accessed with a fixed step or stride between consecutive accesses. Instead of accessing consecutive memory locations, the address of each accessed element is computed by adding a fixed offset or stride value to the previous address.

Consider the following code fragment:

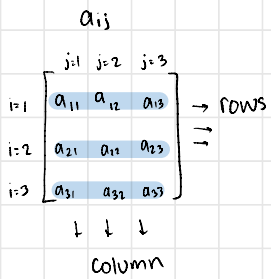
```
1   for (i = 0; i < 1000; i++)
2           column_sum[i] = 0.0;
3           for (j = 0; j < 1000; j++)
4                   column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix `b` into a vector `column_sum`. There are two observations that can be made: (i) the vector `column_sum` is small and easily fits into the cache; and (ii) the matrix `b` is accessed in a column order as illustrated in Figure 2.2(a). For a matrix of size 1000 x 1000, stored in a row-major order, this corresponds to accessing every 1000^{th} entry. Therefore, it is likely that only one word in each cache line fetched from memory will be used. Consequently, the code fragment as written above is likely to yield poor performance. ■

The above example illustrates problems with strided access (with strides greater than one). The lack of spatial locality in computation causes poor memory system performance. Often it is possible to restructure the computation to remove strided access. In the case of our example, a simple rewrite of the loops is possible as follows:

Example 2.6 Eliminating strided access

Strided access patterns can have a significant impact on performance, especially in memory-bound applications
1. Cachec miises: When accessing data with a large stride, the likelihood of cache misses increases because the cache may not be able to prefetch or store the required data in advance. This can lead to additional latency and inefficient memory access.
2. Rediuces Spatial Locality: When data elements are not accessed consecutively but instead with a stride, it introduces gaps in memory access patterns, reducing spatial locality. This can result in less effective utilization of cache lines and decreased cache hit rates, negatively impacting performance.
3. Reduce Memort Bandwidth utilization
4. Increase mamory latency



Alternate Approaches for Hiding Memory Latency

- Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
  - we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
  - we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
  - we access a whole bunch of pages in one go - amortizing the latency across various accesses.

→ Prefetching
→ multithreading
→ spatial locality

Prefetching

↳ used to speed up fetch operations
↳ it involves
  ↳ loading of a resource before it's required to decrease waiting time of that resource

CON
  ↳ might need more space to store advanced loads

Multithreading

↳ run multiple instruction threads at the same time

PROS
↳ concurrent execution of multiple threads
  ↳ allows processor to work on other tasks while waiting for memory operation to complete

Spatial locality

→ as prefetching data

↳ accessing nearby memory locations which can improve
  ↳ cache utilization
  ↳ memory performance

Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become bandwidth bound instead of latency bound.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.

may become bandwith bound

Interconnection Networks

↳ carry data b/w processors and memory

Static (aka direct networks)

↳ links (wires)

↳ links connect nodes together

Dynamic (aka indirect networks)

↳ links + switches

↳ a device which redirects data from any input port to output ports

↳ establish paths among nodes and memory banks

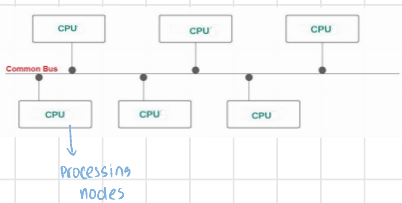
→ Bus Based Network (static)

↳ Processors acess common bus to exchange data

↳ distance b/w 2 nodes is O(1) in bus

↳ Provides broadcast media

↳ scalable cost



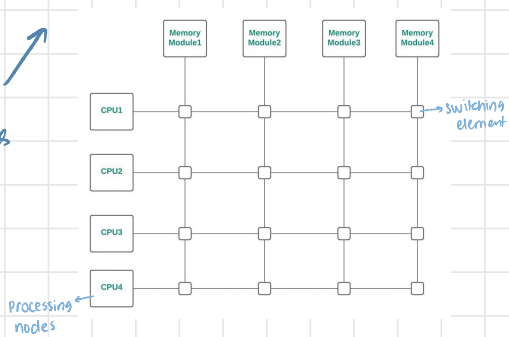
→ Crossbar Networks (dynamic) $O(P^2)$

↳ connects processors to memory banks

↳ a grid of switches

↳ non blocking network (doesnt block other)

↳ scalable performance

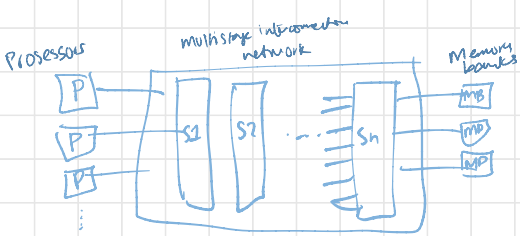


Network Topologies

→ Multistage Network

↳ lies b/w bus and crossbar

↳ more scalable ↔ cost / performance



Interconnection Networks: Network Interfaces

- Processors talk to the network via a network interface.
- The network interface may hang off the I/O bus or the memory bus.
- In a physical sense, this distinguishes a cluster from a tightly coupled multicomputer.
- The relative speeds of the I/O and memory buses impact the performance of the network.

HOW GOOD A NETWORK IS

- diameter ⟶ latency
- no of links ⟶ cost
- degree of nodes ⟶ practicality/cost
- bisection bandwith
  - amount of data that can pass through 2 halves of system

* degree of switch: no of ports