Von Neumann Arch

memory ⟷ CPU

(typically slower than CPU) (high bandwidth)

improvements
- interleaved memory
- caching
- pipelining

Interleaved memory — spread memory addresses evenly across memory banks

└ eg. for two memory banks,
logical address 30 — bank 0
logical address 31 — bank 1
logical address 32 — bank 0

└ ↑ memory throughput
↓ waiting time for memory bank to become available

↓

n-way interleaved
|
n banks

and i^{th} address at $i \bmod n$

Future — multiple processing elements
- Simultaneous multi-threading (SMT) (Hyperthreading)
- Multi core/chip processors (CMP)

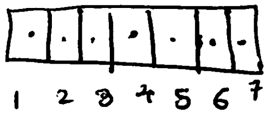
Memory hierarchy
└ registers
caches — which data — spatial and temporal locality

Spatial — stored nearby to recently executed instruction — ↑ chance

Temporal — instruction recently executed is executed again — ↑ chance

Cache associativity

direct mapped — • Cache block can only go in 1 spot in the cache

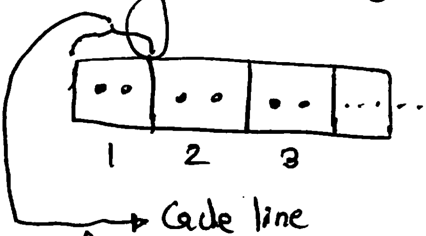


(memory address % cache ~~time~~ size)

• cache block is easy to find

• Not flexible ~~about~~ where to put blocks

2-way set associative — • sets of 2 blocks each



• index – finds set

tag – finds block inside set

fully - associative — • No index – cache block can go anywhere

• Every tag must be compared when finding a block!

• Flexibility of putting the blocks

Cache misses

Cold/Compulsory — block referenced for the first time
└ block doesn't exist yet

Capacity — block not in cache as there is no space in the cache

Conflict — • In set associative and direct mapped

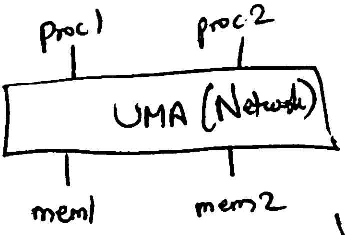
• Multiple blocks can be mapped to a set
└ forcing eviction when set is full

For direct-mapped and n-way associative

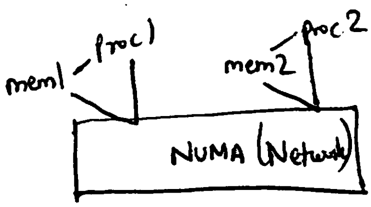
Tag	Index	Offset

Caches have —
- associativity/replacement policy
- block/line size
- Capacity/cache size
- cache line

Parallel machines —
- Bus based: (1) any processor $\xrightarrow{\text{access}}$ any memory location
  - (2) Cheap connection
  - (3) Slow bandwidth
  - (3) Locking mechanism needed
    - L 2 proc accessing same data



- Scalable shared memory machines
  - (1) Uniform memory access
  - (2) Interconnected network with support for remote memory access



- Distributed Memory machines
  - (1) Non-uniform memory access

Paradigms
- Shared memory (shared address space)
- Message passing

"Concurrent" — operations that could be but need not be executed in parallel.

Flynn taxonomy — SISD (Single instruction single data)
— MISD (Multiple instruction single data)
— MIMD and SIMD

Flynn model of computation — machine operates by executing instructions on data

Stream of instructions — tells — "what to do?"

Stream of data — tells — "how will input be affected?"

SIMD algo — flip n coins — calculate no. of heads
→ ① All processors flip a coin
② If coin is head — raise your hand

MIMD — Can work asynchronously
└ different things on different data @ same time

SPMD computing — Single program Multiple Data
① → • Same program run on processors of MIMD machine
• Processors may synchronize

Async parallelism { • Entire program $\xrightarrow{\text{executed}}$ Separate data

Amdahl's law

└── Given n processors, how long would parallel program takes? (same data set assumption)

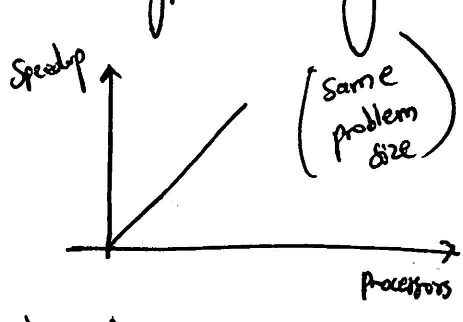
Best achievable speedup is $\lim_{n \to \infty} \dfrac{1}{\begin{pmatrix}\text{serial}\\\text{part}\end{pmatrix} + \begin{pmatrix}\text{parallel}\\\text{part}\end{pmatrix}}$

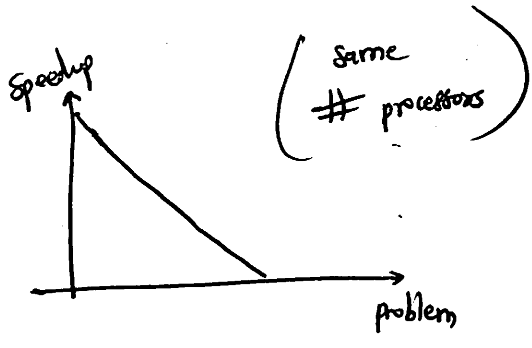
$$= \lim_{n \to \infty} \frac{1}{s + \frac{1-s}{n}} = \frac{1}{s}$$ → for each processor

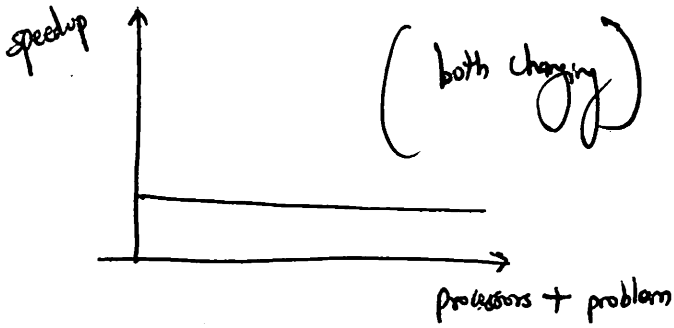
Reality? — Comm^n degrades performance
+ I/O, load balancing, scheduling, etc.

Gustafson's law $\dfrac{1}{s + \frac{p}{n}}$ } Speedup scale @ same rate as n

3 types of scaling







} weak scaling

MPI — for communication among processes
— separate address space

IPC — Synchronization
— Movement of data from one process' address space to another

Message passing — network performance — latency
— bandwidth

Scatter — 'scatter' data items in a message to multiple memory locations

Gather — 'gather' data items from multiple memory locations to one message

Message passing issues

- Naming — how to specify receiver?
- Buffering — what if output is not available?
  Receiver not ready to recv message?
- Blocking — Recv ready to recv before sender ready to send
- Reliability — message lost/in transit?
  Corrupted

Cooperative operation for communication

Push model (active data transfer)

Process 0		Process 1
Send (data)	$\longrightarrow$	Recv (data)

One sided operation for communication (part of MPI-2)

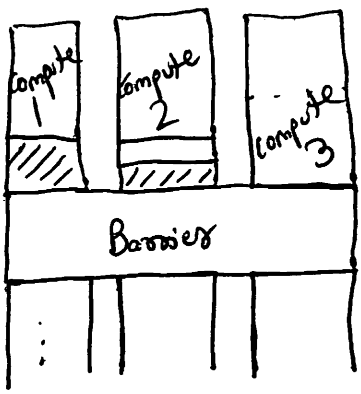
Pull model (passive data transfer)

Process 0		Process 1
Put ~~push~~ (data)	$\longrightarrow$	(memory)
(memory)	$\longrightarrow$	~~pull~~ Get (data)

→ Remote read / writes

Collective Communication (more than 2 processes)

Barrier



One-to-~~all~~ all (Broadcast)

$P_0 \rightarrow P_1$, $P_0 \rightarrow P_2$, $P_0 \rightarrow P_3$

One-to-many (Multicast)

$P_0 \rightarrow P_1$, P_2, $P_0 \rightarrow P_3$

Reduction

$P_1 \rightarrow P_0$, $P_2 \rightarrow P_0$, $P_3 \rightarrow P_0$

↓ one example

MPI — Message Passing Model (an API)

- MPI function returns error codes or MPI_SUCCESS

Two essential env-related questions

- how many processes are participating in this communication?
  MPI_Comm_Size
- Which process am I?
  MPI_Comm_Rank (0 to size-1)

Blocking message passing — Process 0 → May I send? → Process 1: Yes

data transfer + synchronization
(Requires cooperation!!)

Data in MPI - message - (address, count, datatype)
where datatype can be MPI_INT, MPI_DOUBLE, etc.
also custom datatypes

How to identify message — tags
- specific tag (like number)
- MPI_ANY_TAG (for any tag) — at recv side

$$\text{MPI\_Send}(\underbrace{\text{start, count, datatype}}_{\text{message buffer}}, \underbrace{\text{dest}}_{\text{target process}}, \underbrace{\text{tag}}_{\text{identification}}, \text{comm})$$

blocking

comm → usually

MPI_COMM_WORLD – default communicator whose ranks contain all initial processes

When it returns –
- data is delivered
- buffer can be reused
- message MAY NOT have been received by target process

$$\text{MPI\_RECV}(\text{start, count, datatype}, \underbrace{\text{source, tag}}_{\text{waits for a matching on source and tag}}, \text{comm, status})$$

status → further info

{ Receiving fewer than 'count' OK!
{ Receiving more than 'count' ERROR!

using 'Status' → Status.MPI_TAG – recvd tag

Status.MPI_SOURCE – recvd source

MPI_Get_Count() – no. of recvd elements

Blocking doesn't mean — message was delievered to recv/destination

it means — send/recv buffer is available for reuse

A blocking send can complete as soon as message was buffered, even though no matching receive has been posted

However, message buffers can be expensive!

So MPI provides modes —

- Standard mode — upto the library — whether or not to buffer the outgoing message
- Buffered — if 'send' started and no matching 'recv' posted — outgoing message must be buffered.
- Synchronous — send can be started whether or not matching recv is posted
  — recv has started to recv message

  The completion of send — buffer can be reused
  — recv process has started to recv data
- ~~B~~ Ready — Unlike other 3, send can be started only if matching recv is posted.

Blocking

- do not return till commⁿ is finished
  - buffer passed to MPI_Send() can be reused
  - MPI_Recv() returns when the recv buffer has been filled with valid data
- Using MPI_Send() and MPI_Recv()

Non-blocking

- These fⁿs return immediately
  - even if commⁿ is not finished
- We need to call MPI_Wait() or MPI_Test() to check if commⁿ has finished.
- Using MPI_ISend() and MPI_IRecv()
- We can do MPI_ISend() then some computations, then MPI_Wait()
  - faster performance

MPI_Comm_Split — to create new communicators

Collective operations
- MPI_[ALL] SCATTER [V]
- MPI_[ALL] GATHER [V]

Collective operations are called by all processes in a Communicator

No tags

Blocking

MPI_BCAST — distributes data — one process ⟶ all others in a communicator

MPI_REDUCE — Combines ~~distributes~~ data — all processes in a Communicator ⟶ one process

⟶ predefined ops — MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc.

MPI_ALLREDUCE — Combines values back from all processes and distributes ~~#~~ result back to all processes

Syntax

MPI_Bcast (sendbuf, count, datatype, rank of root, comm)

MPI_Reduce (sendbuf, recvbuf, count, datatype, op, rank of root, comm)

op ⟶ reduce Operation like MPI_SUM

Source of deadlock

→ send a large message from process 0 to process 1

→ if insufficient storage at destination,
send must wait for user to provide memory space

	Process 0	Process 1
can cause deadlock	Send(1)	Send(0)
	Recv(1)	Recv(0)

Solution — Use non-blocking operations

	Process 0	Process 1
	Isend(1)	Isend(0)
	Irecv(1)	Irecv(0)
computations can happen here	Waitall	Wait all

~~Synchronization~~ Why efficient —

- prevents deadlocks
- improves performance by allowing overlap of communication and computation
- avoids overheads of allocating buffers and copying messages to buffers.

More optimization

Process 0	Process 1
Irecv (1)	Irecv (0)
Isend (1)	Isend (1)
Waitall	Waitall

Why? As data can be moved directly to the recv buffer and there is no need to queue a pending send request.

Another alternative to avoid deadlock
- Use different communicators
- Check out Multicasting (if applicable)

MPICH — "high performance portable implementation of MPI (1+2)"
(alternative: Open MPI)

PMPI — profiling layer of MPI

example of flow
```
MPI_Init() { //wrapper
  //pre stats
  PMPI_Init();
  //post stats
}
```

When to use MPI?

- need to manage memory on per processor basis (No shared memory concept here)
- portability
- performance
- Irregular data structures (due to MPI_datatype being dynamic) — interesting

MPI does not work well for

- fault tolerance
- distributed computing
- embarrassingly parallel data division like Google's map-reduce

MPI-2

- Parallel I/O
- One-sided operations
- Dynamic process management (Spawn new processes at runtime and communicate between them)

One-sided communication

non blocking

- MPI_Put – stores to remote memory
- MPI_Get – reads from remote memory
- MPI_Accumulate – like Reduce
  - we need "op" here like MPI_SUM

GPGPU – General Purpose GPU

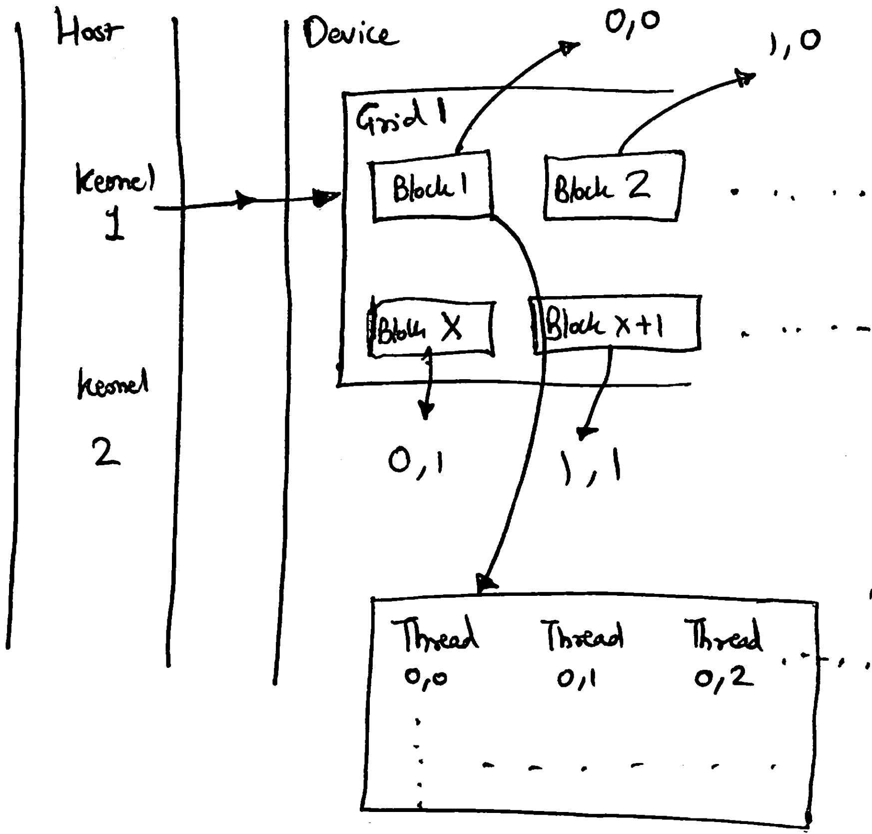
- low latency FP computation
- fine grained SIMD parallelism
- large data arrays

GPU has more transistors for computation (instead of caching or flow control) and is more suitable for data-intensive stuff

CUDA – Compute Unified Device Architecture

- contains drivers for loading computation programs to GPU
- GPU is treated as a coprocessor to CPU
- has its own DRAM (device memory)
- Runs threads in parallel
  - Overprovision of threads hide latencies — also very lightweight
- Data parallel portions — executed on kernels (devices) runs parallel on many threads

Single cycle context switches provide latency hiding



Can cooperate with each other (have shared memory)

	Executed on the:	Only callable from the:
Device func()	device	device
Kernel func()	device	host
Host func()	host	host

Each thread can —
- read/write per thread registers
- read/write per thread local memory
- read/write per block shared memory
- read/write per grid global memory
- read-only { per grid — constant memory — texture memory

Host can read/write these 3 memory spaces

Constant memory — effective when all threads access the same memory at the same time
(rather small)
→ reduces memory bandwidth

Texture memory — for patterns exhibiting a high deal of spatial locality
(quite large and has its own cache)

some config info

```
dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
kernelfunc<<< DimGrid, DimBlock, SharedMemBytes>>>();
```

kernelfunc → they are asynchronous

```
idx = (blockIdx.x * blockDim.x)
      + threadIdx.x;
```

`__syncthreads()` — block level synchronization barrier
→ safe to be used when all threads in block reach same level

How to sync all threads in grid?
└→ use consecutive kernel calls
(as all threads end and start again from same point)
↓
CPU synchronization / implicit synchronization

Extensions — Open ACC

Multiprocessor can execute multiple blocks concurrently
└→ Shared memory and registers are partitioned among threads of all concurrent blocks.

Threads, Warps, Blocks
└→ Upto 32 threads in a warp
Upto 32 warp in one ~~core~~ multiprocessor
16 or 32 such multiprocessors – more is better
16 (atleast) blocks required to fill the device

Nice summary

device = GPU = set of multiprocessors

Multiprocessor = set of processors and shared memory

Kernel = GPU program

Grid = array of thread blocks that execute a kernel

Thread block = grp. of SIMD threads that execute a kernel and utilize shared memory

Local memory (for 1 thread) is off-chip	Shared-memory (for all threads in a block) is on-chip

memory spaces

- Local / Global — not cached — slow
- Texture / Constant — cached

```
__shared__ int scratch[blocksize];
~~Scratch[blocksize]~~
scratch[threadIdx.x] = begin[threadIdx.x];
//Compute on scratch values
begin[threadIdx.x] = scratch[threadIdx.x];
```

Scratchpad memory

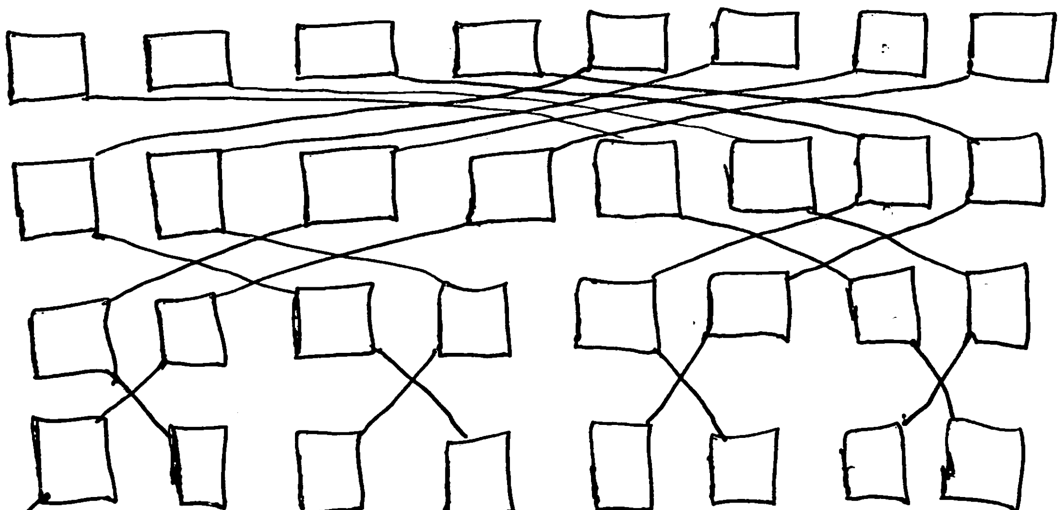
```
Scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

Communicating values between threads

Parallel reduction using butterfly pattern (log n steps) (for n threads)

each step holds 1 element

stepwise partial sum



every thread now hold sum in sum[i]

Butterfly pattern algorithm

```
int i = threadIdx.x;
-shared- int sum[blocksize];
sum[i] = x_i; _syncthreads();
for (int bit = blocksize/2; bit > 0; bit /= 2) {
    int t = sum[i] + sum[i^bit];
    _syncthreads();
    sum[i] = t;
    _syncthreads();
}
```

OpenACC — execution constructs
OpenACC — data constructs

more implicit
- kernel — runs kernels on GPU (1 kernel / loop)
  do not block when done — ASYNC

programmer specifies how to parallelize
- parallel — Run 1 kernel on GPU
- wait — barrier
- loop — run iterations of loop on GPU

architecture naming
- Gang → SM (thread block)
- Worker → Warp
- Vector → Thread

OpenACC also has data caching
└ uses shared memory

#pragma acc cache( )

SMPs (Shared Memory Processors)
└ OpenMP — shared memory parallelism (an API)

Another abstraction (or high-level) — Threads
└→ • Threads of a process share same address space and same resources
  • Supports SPMs : # threads = # processors
  • Each thread contains — execution state, execution context (registers), and a per-stack thread,

So the stack space is divided by each thread

Sync is necessary in threads
└→ As although they have different stack space, they share the heap and globals.
  └→ Data integrity must be maintained

So we need Mutual Exclusion

Race conditions are usually defined in terms of threads
└→ potential for interleaved execution of a critical section by multiple threads
  └→ Causing non-deterministic results

Producer / Consumer example — for circular queue

Producer

```
Mutex_lock ( );
while (count == max) {
    Conditional_Wait (E);
}
Put ();
Conditional_Signal (F);
Mutex_Unlock ( );
```

tell others sleeping on the conditional variable to wake up

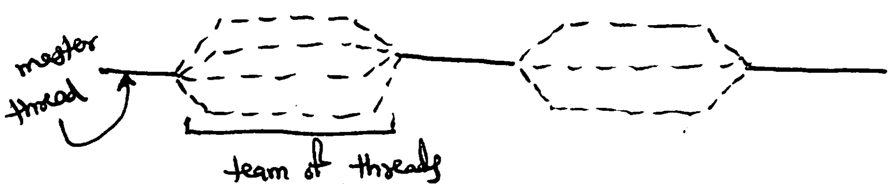
~~Consumer the~~ sleep till count != max

Consumer

```
Mutex_lock ();
while (count == 0) {
    Conditional_Wait (F);
}
Get ();
// Some Calculations
Conditional_Signal (E);
Mutex_Unlock
```

OpenMP model ("fork-join parallelism")

→ Master thread is spawning team of threads as needed.



How to extract performance from OpenMP:

- OpenMP is shared memory model → threads comm by sharing variables → But there can be race conditions → So we need synchronization → But it is expensive to sync
- → Avoid/Minimize need for synchronization

OpenMP constructs fall into 5 categories —

1. Parallel regions
   - create threads with "omp parallel"

```
#pragma omp parallel {
    // parallel region's code
}
```

→ a single copy of the input data (if any) is shared between all threads

→ threads will wait here for all threads to finish before proceeding — implicit barrier

2. Work Sharing
   - "omp for" — splits loop iterations among threads in a team

Optimization trick

We can remove the barrier — using nowait
   - Useful when we have two consecutive independent for loops

Scheduling — • static — iteration blocks are mapped statically to the execution threads in a round-robin fashion.

Advantage — improves locality in memory access
Disadvantage — something like static (schedule, 2)

might worsen the performance

- Dynamic — works on "first-come first served" basis.
  Two runs with the same number of threads
  might produce completely different iteration space.
  $\underbrace{\hspace{20em}}_{\text{helpful when computation times vary}}$

- Guided — ① dynamic grabbing of blocks of iteration
  ② We start large and shrink down to a chunk
  $\underbrace{\hspace{20em}}_{\text{reduces scheduling overhead seen in dynamic}}$

Optimization Caution { Make sure multiple threads do not overwrite each other's variables

3. Data Environment

Most variables are shared by default

Not shared ones (Private var) — stack variables from parallel regions
— loop iteration variables

(But only 1 loop iteration var is allowed!)

Private clause 'nuance'
└ creates a local copy for each thread

What can go wrong

```
void wrong() {
  int IS = 0;
  #pragma omp parallel for private (IS) {
    for (j=0; j<100; j++) {
      IS += a[j];
    }
  }
  printf("%d", IS);
```

I Error! As the value is uninitialized in #pragma part

error → IS is undefined at this point! II

"First private" solves I but II remains

"Last private" solves II but I remains
└→ it is the last iteration's value

as iterations are broken down, this might be partial sum and not what we want

~~Looks like only Fortran APIs support private~~
└→ Not true!

threadprivate (A) — each thread has its own copy of A
— Global (A) has same values before/after (No II)

difference
- Thread private ends up on stack
- Copy private ends up on heap

Reduction — Syntax — reduction (op : list)

operation

the variables in "list" must be shared in parallel region

for example - last page code - reduction(+ : IS)

Solves BOTH I and II

they have initial value like

+ → 0

* → 1

4. Synchronization

i. #pragma omp critical {
      consume ( );
   }

→ threads wait their turn and call the critical section one at a time (for the consume ( ) call)

ii. for update of memory location, use

#pragma omp atomic {
      X = X + temp;
}

iii) #pragma omp barrier

→ each thread waits untill all the other threads arrive

iv) #pragma omp ordered

→ enforces sequential order for the block

Some misc work sharing constructs

#pragma omp master

→ block is only executed by the master thread

→ No barrier implied at the end of it

#pragma omp single

→ block is only executed by a single thread

→ barrier implied at the end!

Collapse (# of loops to collapse)

- require body independent of respective loops
- no statements in the outer loops allowed

#pragma omp for simd

→ No compiler correctness check!

Vectorize instructions (no conditionals allowed in loop body)

Open MP 4.0 → support for host to device transition

Cache and Memory Systems

Flops – floating point operations

→ + × – 1 flop

/ or sqrt – 4 flops

log – 8 flops

compare – 1 flop

} # of flops for an operation

CPU time vs Wall clock time

CPU time: Scheduling overhead, Communication overhead

Wall clock time: preferred

Hardware performance counters

↓

Statistics →

- cycle count / load/store count
- instruction count
- branch taken / not taken count
- branch misprediction count
- Cache invalidations
- pipeline stalls

PAPI – to instrument applications

Time to run code = clock cycles running code + Clock cycles waiting for memory

Clock cycles waiting for memory: major bottleneck!

Also, I/O access has the highest latency

(~ 5–15M cycles)

↓

L1 has 4 cycle latency

Tuning for caches

- Preserve locality
- Reduce cache thrashing
  ↓
  use split caches
  - data
  - instruction
- loop blocking when out of cache

```
for (i=0; i<N; i++) {
}
        ⇓
for (j=0; j<N; j+=B){
  for (i=j; i<min(N, j+B);
       i++){
  }
}
```

Optimal cache lines — depends on data bus / memory subsystem

L1 misses are handled (2x–10x) times faster than L2's.

Policies for line-replacement
- random (cheap/simple)
- LRU (preserves temporal locality / expensive)
- FIFO (for superior)

Write through — all writes update cache and underlying memory / cache
(valid bit) Simpler management of cache

Write-back — all write update cache
(valid bit and dirty bit) may have to write back to memory
lower bandwidth

→ Write policy

When a write miss
— do not allocate new cache line
go through to underlying memory / cache
— allocate new cache line
read! — expensive

Cache trashing — frequently used cache lines replace each other

↓

reasons

→ indirect addressing (sparse matrices)

→ very large array

→ Instruction and data conflict (for unified caches)

- Loop unrolling — increases instruction parallelism
- Prefetch data to reduce misses

$\boxed{\text{loading}}$

→ binding prefetch (to register)

→ non-binding prefetch (to cache)

Code optimizations

1. Merge arrays

before
```
int val[size];
int key[size];
```

after
```
struct merge {
    int val;
    int key;
}
struct merge merge_arr[size];
```

- Reduces conflicts between val and key
- Improves spatial locality

2. loop interchange

```
for (k=0...)
for (j=0...)
for (i=0...)
    x[i][j] = 2 * x[i][j]
```

```
for (k=0...)
for (i=0...)
for (j=0...)
    x[i][j] = 2 * x[i][j]
```

- Improves spatial locality

3. loop fusion

```
for i
    for j
        a[i][j] .....

for i
    for j
        b[i][j] .....
```

```
for i
    for j
        a[i][j] .....
        b[i][j] .....
```

- Improve temporal locality

- loop blocking

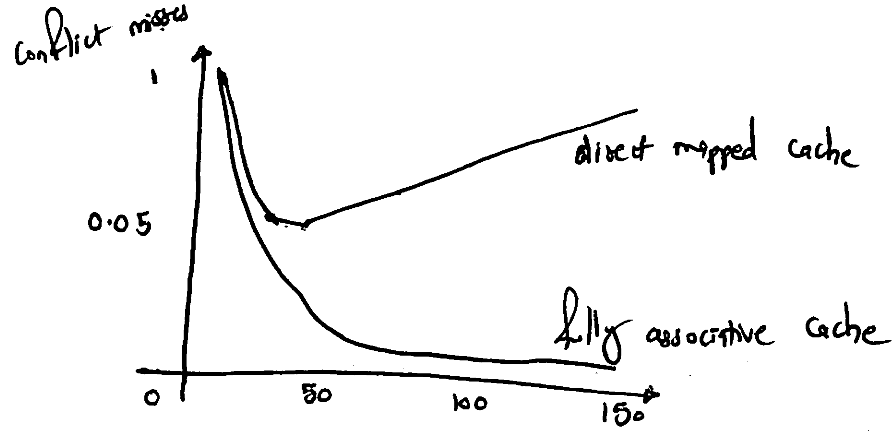
(example showed before)

B was the blocking factor

It can change a 3-for loop from N^3 to N^3/B

multi level tiling (typically best for L1 and L2)

single level tiling (typically best for L1)



performance optimization: If things are mapped to the same cache line, use padding!

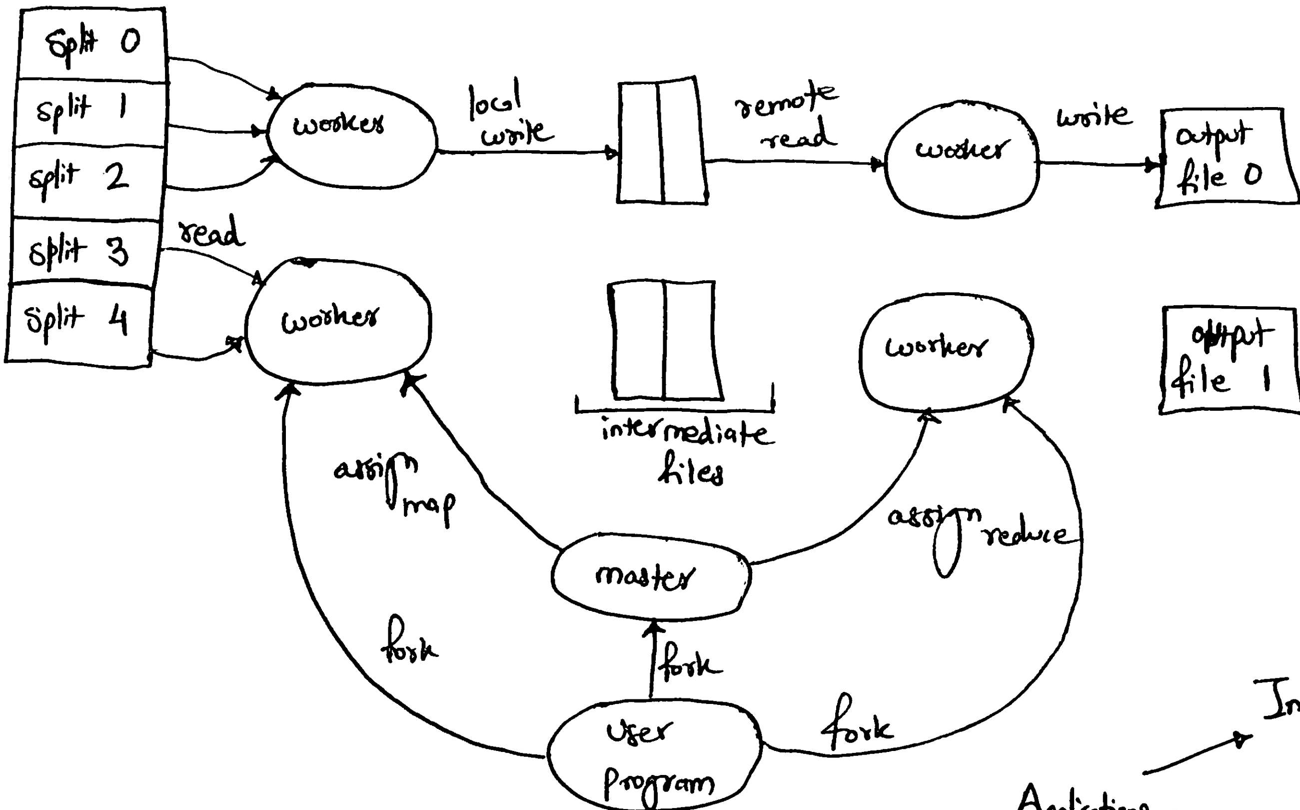
Average memory access time = (Hit time) + (Miss rate * Miss penalty)

Map reduce – processing and generating large data sets

→ programs automatically ∥ → executed on commodity machines connected together with switched ethernet

→ an abstraction

→ LISP → map/reduce primitives



Applications → Inverted Index
Applications → Term-vector per host
Applications → distributed grep
Applications → Count of URL access frequency
Applications → Reverse-web link graph

$$\text{Map} \rightarrow (k_1, v_1) \longrightarrow \text{list}(k_2, v_2)$$

$$\text{Reduce} \rightarrow (k_2, \text{list}(v_2)) \longrightarrow \text{list}(v_2)$$

Why replication → availability

Why replication → reliability

Map – M splits

Reduce – R splits } the intermediate key space is distributed into R pieces using $\underbrace{\text{hash(key) mod R}}_{\text{fairly well-balanced schedules}}$

Mapreduce frameworks handle the I/O for us

1. Split input files → M pieces → 16 MB to 64 MB per piece
2. Starts many copies of program on a cluster of machines
3. Master → picks up idle workers → assigns each one a map/reduce task
4. Worker with map → read input split → parse key/value pairs from input data → user-defined map function → intermediate key/value pairs → buffered in memory
5. Periodically, buffered pairs are written to local disk → locations are passed to master → forwards to reduce workers
6. Reduce worker $\xrightarrow{\text{RPC}}$ read data from local disk → sorts by intermediate keys (same key are grouped together) — may use external sort (depends on memory size)
7. Reduce worker → iterates → sorted intermediate key/value → for each unique key → with all corresponding values → user's reduce function

Master stores
1. state → idle, in-progress, completed
2. identity of worker machine

Map-reduce is bad for →
- jobs with shared state or coordination
- low-latency jobs
- small datasets
- finding individual records

Worker failure

↳ master pings worker periodically $\xrightarrow[\text{response}]{\text{no}}$ "worker failed" marked by master

↓ map/reduce tasks

eligible for rescheduling

Master failure → maintains periodic checkpoints → if died, copy starts from last checkpoint state

Mapper and Reducer should be stateless — they should remember ~~nothing~~ nothing about processed data

↳ as there is no guarantee which key/value pair will be processed by which worker

Notes while coding → 1. Mapper and reducer should be stateless
2. Do not try your own I/O
3. Mapper must not map too much data to the same key

Locality → master attempts to schedule map task on machine containing replica of input data

↳ if failed to do that → schedule near a replica of that task's input data — like on a machine that is on the same network switch as the machine containing the data

maintained by GFS (google file system)

M, R > no. of worker machines → improves dynamic load balancing
→ speeds up recovery when a worker fails

Master makes O(M+R) scheduling decisions

keeps O(MR) state in memory

Realistically → M — divide such that individual task is roughly 16-64 MB of input data

R — small multiple of the number of worker machines

Optimizations — 1. 'Straggler' — machine that takes unusually long time to complete one of the last few map or reduce tasks in computation.

eg: bad disk

How to overcome this? — when close to completion, master schedules backup operations of the remaining in-progress tasks

→ completed when backup or primary execution completes.

2. Combiner → partial merging of data before sent over the network

→ on a machine that performs map task

3. Acceptable to skip a few records

→ framework detects records which deterministically crashes

→ skips them

3. Continued - how skipping happens?

signal handler sends a 'last gasp' UDP packet to master
- contains sequence number
  - when master has seen the record fail many times, it instructs map/reduce tasks to skip it

4. Counters - to count occurrences of various events
- master also eliminates duplicate counting
- workers piggyback counts to master via a ping response

Expanding upon worker failure
- i. Re-execute completed and in-progress map tasks,
  even completed tasks have to be rescheduled as the output is on local disks → inaccessible
- ii. Re-execute in progress reduce tasks
  not required for completed as output is stored in a global file system

Hadoop Distributed File System → stores metadata on a dedicated server - namenode
→ application data stored on other servers - datanode

Servers are fully connected and communicate using TCP-based protocol

Unlike Lustre, HDFS does not use data protection mechanisms like RAID
- like GFS, file contents are replicated on multiple data nodes for reliability

Trend → distributed implementations of namespace

e.g.- Ceph — cluster of namespace servers (MDS) and

dynamic subtree partitioning algorithm → namespace tree $\xrightarrow{\text{map}}$ MDS evenly

HDFS — hierarchy of files and directories represented on namenode by inode — permissions, modification, access time, namespace, disk space quota } metadata

replication { file $\xrightarrow{\text{split}}$ large blocks (typically 128 MB) $\xrightarrow[\text{block}]{\text{each}}$ replicated at multiple datanodes (typically 3)

HDFS keeps namespace in RAM

read/write flow {

HDFS client $\xrightarrow{\text{contacts}}$ Name node for location of data blocks $\xrightarrow{\text{reads}}$ block contents from datanode closest to the client.

HDFS clients $\xrightarrow[\text{data}]{\text{writing}}$ requests Namenode to nominate 3 datanodes to host block replicas $\xrightarrow{\text{writes}}$ in a pipeline fashion to 3 datanodes

Terminology {

inode data + ~~metadata~~ list of blocks belonging to each file } metadata $\xrightarrow{\text{called}}$ image

persistent record of image $\xrightarrow{\text{in}}$ local host's native filesystem $\xrightarrow{\text{called}}$ checkpoint

Namenode $\xrightarrow{\text{stores}}$ modification log of image $\xrightarrow{\text{called}}$ journal

terminology (continued)

Block replica on data node $\xrightarrow{\text{represented}}$ two files → data
→ block's metadata like checksums and generation stamp

for HDFS { size of data file == actual length of the block

Handshake → during startup → datanode connects to namenode for handshake

verifies namespace ID and software version of datanode

after handshake → datanode $\xrightarrow[\text{with}]{\text{registers}}$ name node

and namenode assigns internal identifier — Storage ID
(recognizable even if restarted with different IP address or port)

How are block replicas identified? → datanode sends block report to namenode (every hour)

block report → block ID, generation stamp, length

Heartbeats { datanode $\xrightarrow{\text{heartbeat}}$ namenode } default interval — 3 seconds

If no heartbeat in 10 mins → datanode out of service → block replicas hosted by datanode become unavailable

Heartbeats
- helps in name nodes' space allocation and load balancing decisions
- sends
  - total storage capacity
  - fraction of storage in use
  - number of data transfers in progress

how comm^n happens between datanodes and namenodes?
1. Namenodes does not directly call datanodes
2. It uses replies to hearbeats to send instructions to datanodes

Secondary Namenode $\xrightarrow{\text{holds}}$ backup of the namenode data

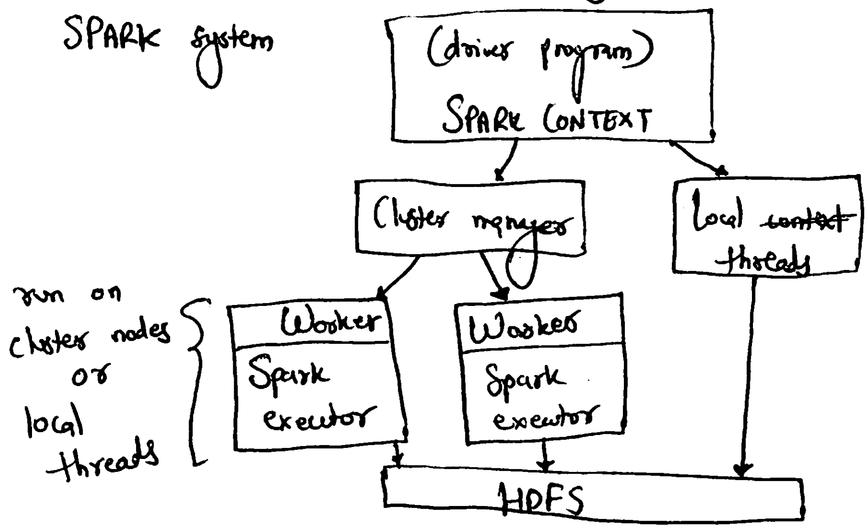
Block placement → impractical to connect all nodes in flat topology
- → spread nodes across multiple racks
  - → nodes of racks share a switch

Multithreaded nature of name node
- Namenode is multithreaded system
  - → processes requests simultaneously from multiple clients
- bottleneck → flush and sync procedure $\xrightarrow{\text{mitigated}}$ name node batches multiple transactions initiated by different clients → when one calls flush-and-sync → all transactions batched at time T are committed together

Backup node
- creates periodic checkpoints
- maintains in-memory, up-to-date image of file system namespace → sync with Namenode

Apache Spark
- not modified version of Hadoop
- in-memory data storage — x40 faster than Hadoop
- map-reduce like engine with fast iterative queries

SPARK system



Why Spark?
→ for faster data sharing across ff jobs
as we are replacing disk I/O with distributed memory

RDD — Resilient Distributed Datasets (RDDs) — distributed memory abstraction
→ for in-memory computations on large clusters in a fault tolerant manner
→ provides restricted form of shared memory
→ supports node failure and recovery

How to build RDDs
→ parallelize existing collection in driver program
→ referencing dataset in external storage system (like HBase, HDFS)

To parallelize existing collection — how many partitions? to cut the dataset into — Spark runs one task for each partition of the cluster

Typically 2-4 / CPU in a cluster

RDD supports two types of operations
— transformations (lazy) — create new dataset from an existing one
— action (immediate) — return a value to the driver program after running computation on dataset

transformation ⟷ map

action ⟷ reduce

also 'reduce by key' — returns distributed dataset

Optimizations
1. Transformations are lazy → computed when action requires result to be sent to driver program

2. Persist an RDD in memory

For fault tolerance
1. RDDs maintain lineage information → to reconstruct lost partitions
2. While distributed memory allows reads and writes to each memory location, RDDs are restricted to bulk writes → also no overhead of checkpointing

transformation examples
- filter (f^n) — returns new dataset → where f^n returns true
- flatmap (f^n) — each input item $\xrightarrow{\text{mapped}}$ 0 or more output items

Two RDD transformation on RDDs — union, intersection, substract, cartesian

action examples
- collect () — returns all the elements of the dataset as an array at driver program
- count () — no. of elements in the dataset
- take(n) — returns first n elements of the daset — as an array

RDDs can be stored as
- memory | memory and disk — deserialized Java objects
- memory | memory and disk — serialized Java objects (one byte array per partition) more space efficient but more CPU intensive to read
- disk only

Spark runs a f^n in ~~||~~ as set of tasks $\xrightarrow{\text{ships}}$ copy of each variable in the f^n to each task

for variables which need to be shared across tasks
- broadcast variable (cache a value in memory on all nodes)
- accumulator (variables that are only added to, like counters and sums)

TensorFlow — tensors — data + transformations

Keras — high level neural networks API

PyTorch — GPU based tensor library

Dense layer — each neuron connected to all in previous layers
└→ CNN — only connects to a small local set of neurons

image → nxn weights kernel convolutional → feature map $\xrightarrow[\text{them}]{\text{stack}}$ single tensor
output of a convolutional layer

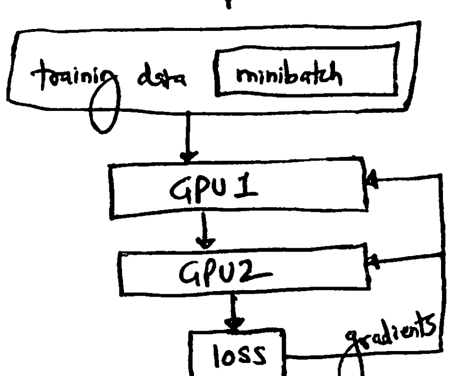
color images have 3 color channels — red, green, blue

To reduce spatial resolution and complexity and number of parameters
└→ Pooling layers → like MAX, AVERAGE

ML, GPU - Parallelism

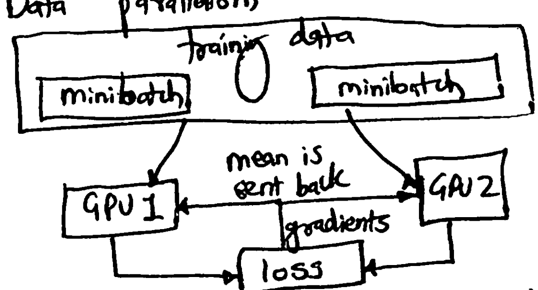
Nvidia (Volta + Ampere) — Tensor cores ← specialized hardware for tensor multiplication
uses half precision (FP16) (16 bits)
useful when high precision is not essential)

Model parallelism



① over independent layers (limited parallelism)
② If using same layers → bad!
└→ Outdated weights due to dependency

Data parallelism



① over batches (GPUs, nodes)
② all / to all exchange weights
↑ This is more common

Horovod uses it

Nvidia NCCL — Collective communications library — multi GPU, multi node comm primitives
└→ provides routines all gather, all reduce, broadcast, reduce, reduce-scatter
point-to-point send and receive
high bandwidth low latency over PCIe and NVLink

Slurm — workload manager — uses a best fit algorithm based on Hilbert curve scheduling

Single-node, multi GPU data parallelism — tf.distribute.MirroredStrategy()

└→ batch size is split among GPUs (batch_size/gpu of data each GPU gets it)

complexity

keras, TF

- model.fit() — quick experiment
- model.fit() + callbacks — add checkpointing, early stopping, TensorBoard monitoring, etc.
- model.train_on_batch() + callbacks — with built-in optimizers and losses
- custom training loop with gradient type — can provide new optimization algo

Sequential() — code runs sequentially on CPU with numerical part on GPU

Callbacks — for invoking user methods — like for checkpointing

MirroredStrategy() —
1. for 2+ GPUs per node
2. Synchronous data parallelism
3. Variables mirrored on each GPU
4. Variable creation should be done within strategy's scope
   └→ replicate variables across all replicas and keep them in sync using all-reduce

add checkpoints at epochs via callbacks
↓
gives fault tolerance

MultiWorkerMirroredStrategy() —
1. for 2 nodes with 1+ GPU/node
2. Synchronous data parallelism
3. Variables mirrored — replicas coordinate using all-reduce

all-reduce is at batch granularity

parallel-IO { Multiple processes of a parallel program accessing data from a common file

→ using individual file pointers (MPI_file_seek, MPI_file_read)

→ using explicit offsets (MPI-File_read_at)

same for write — MPI_file_write, MPI_File_write_at

Non-blocking I/O → MPI_File_iwrite_at, MPI_Wait } recall: — f^n returns immediately, even if comm^n has not finished, we can do some computations before MPI_Wait()

Non contiguous access + Collective I/O } highest performance

MPI I/O provides this within a single function call using derived datatypes like vector, indexed block, struct, subarray, contiguous, etc.

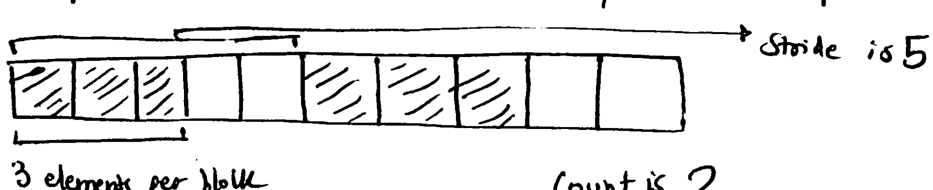
There is also → file views → MPI_File_set_view — assigns regions of the file to separate processes

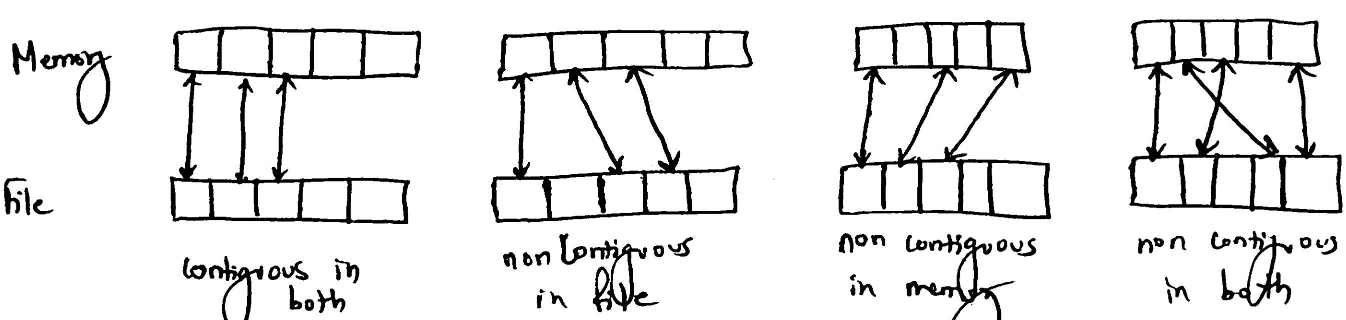
(displacement, → number of bytes to be skipped from the start of the file

etype, → basic unit of data access

file type) → portion of the file accessible/visible to this process

Vector datatype →





HPC I/O patterns → N files — each rank has own file (high directory contention)

→ 1 file — each rank has own offset (high file contention) (high block bound contention)

→ Hybrid — k files shared by m ranks (low contention for file, directory and block bounds)

optimal

Collective I/O —

- each process may need to ~~place~~ access several non contiguous portions of the file, but together, it might be a large contiguous position of the file.

  combining multiple processes

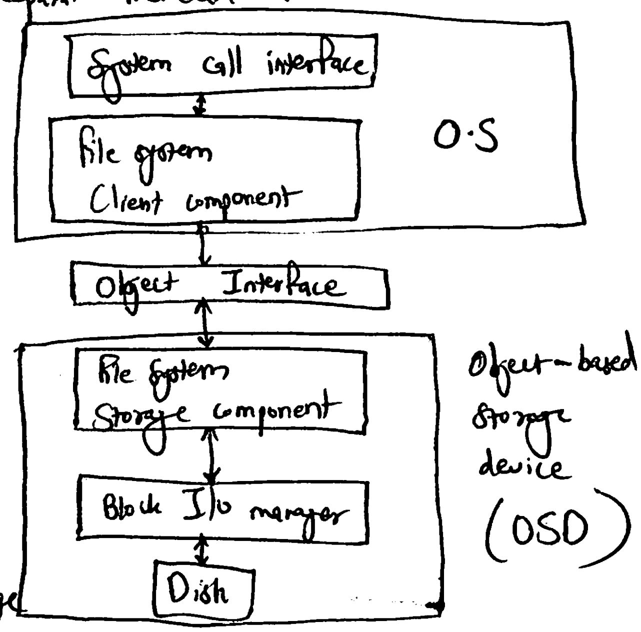
- Collective I/O merges requests of different processes and services the merged request.
- MPI_file_read_all, MPI_File_write_all

HPC data formats

- POSIX — read/write basic API
- MPI-IO — ~~//~~ read/write API
- HDF5 — hierarchical data format
- NetCDF — Network common data format

Ceph — object-based storage devices and file systems

separate metadata and file data



key ideas — ① Object-based storage

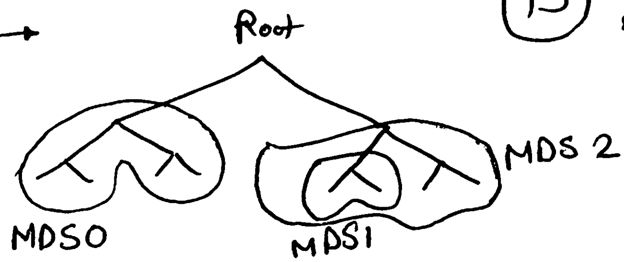
② Decoupling metadata and data

metadata server — OSD

They use CRUSH (Controlled replication under scalable hashing) — to calculate how and where to store and retrieve data — distributed object based storage

15

Dynamic subtree partitioning →



adaptively distributes cached metadata across a set of nodes

Distributed Object Storage → files $\xrightarrow[\text{across}]{\text{split}}$ objects $\xrightarrow[\text{form members of}]{\text{are combined to}}$ placement groups

placement groups $\xrightarrow{\text{CRUSH}}$ assigned to an OSD $\xrightarrow{\text{group by}}$ failure domain

$$\underbrace{CRUSH(x) \rightarrow (osd_1, \underbrace{osd_2, osd_3}_{})}_{\text{cluster map infrequently updated}}$$

→ replication is happening here

Lustre → MDS
→ OSS (object storage servers)
→ Lustre clients
→ OST (object storage target) → software component
→ lives in OSS
→ interfaces to backend volume

Stripe count — # of OSTs used to store the file

recommendation default stripe size is 1MB { if one large file → $\underbrace{\text{Stripe over all OSTs}}_{}$ so that work is equally distributed among the clients

if large number of files (~2 times # OSTs) → stripe count = 1

Main Recommendation { Stripe $\boxed{\text{enough}}$ OST to keep all OSTs busy on read and write paths

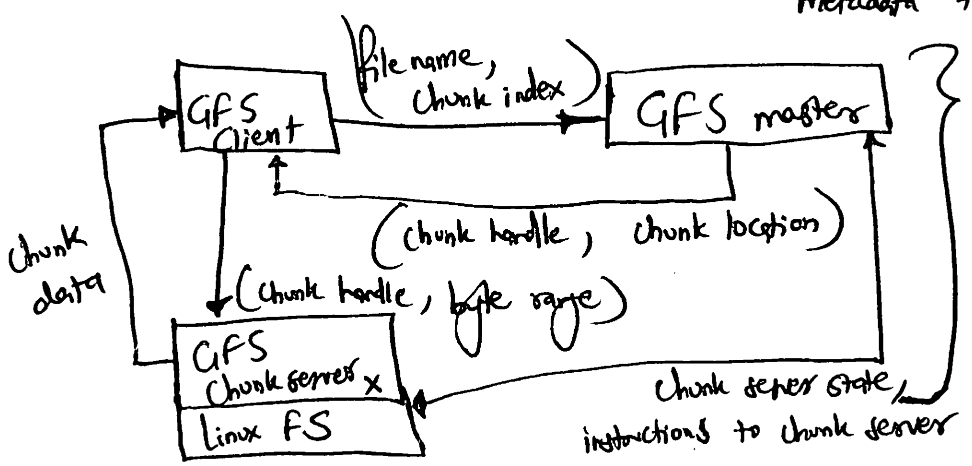
Google File System → commodity hardware
→ modest # of large files (each ~ 100 MB to multi-GB)
→ read-mostly workload
→ high throughput, low latency
↓
reliability through replication

design decisions:

- GFS stores replicated chunks on linux File system as local files
- single master per GFS cluster
- Periodic heartbeats to checkup on servers
- No caching } just rely on linux buffer cache — Why? large data transfer

GFS master $\xrightarrow{\text{stores}}$ metadata (64 bytes per 64 MB of data)
→ file/chunk namespace, mapping, location of chunk and replicas
→ in Memory
→ stores checkpoints of an operation log, recovery

GFS chunkserver → 64 MB, large chunks — this reduces metadata overhead, as less metadata to store



} read flow GFS → two types → large streaming reads; → small random reads

side-topic map-reduce paper talks about leases in GFS { HDFS file — granted a lease (for writes) — renewed via heartbeats

soft limit: till it expires, writer is certain of exclusive writes; after expiry, another writer can preempt

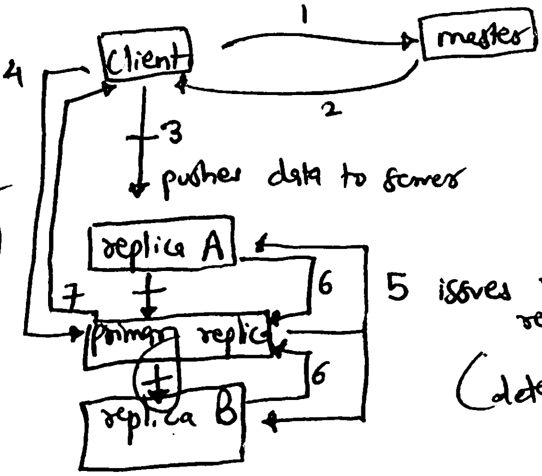
hard limit: after it expires (one hour), HDFS assumes client has quit — closes the file for the writers

Note { Clients can ask for multiple chunk locations in a single request
AND
Polling of chunk servers at startup takes place

17

GFS $\underbrace{\text{mutation}}$ → any write or append

issues write to primary



5 issues write to replicas (determines the order)

(lease duration here is 60 seconds)

Snapshot — creates copy of file or directory at low cost — $\underbrace{\text{copy on write techniques used}}$ like AFS

Locking — lock per path — to access $\underbrace{\text{/d1/d2/abc}}$

lock /d1, /d1/d2, /d1/d2/leaf

each thread — read lock on directory, write lock on file

for fault tolerance { Master replication — shadow masters provide read-only access when primary masters are down

Performance — why no linear time speedup — algorithm limit / bandwidth limit / architecture limit

Why amdahl's law is not reality? — parallelization creates overhead

- interprocess communication and synchronization
- Idling (waiting) (due to load imbalance, serialization, resource contention)
- excess computation

Usage charging policy (like for AWS) { SUs = # processors allocated * wall time * priority level

metrics {

① Total parallel overhead $= pT_p - T_s$

② Speedup $= \dfrac{T_s}{T_p}$ (if $> p$ then super-linear speedup)

T_p – parallel execution time

T_s – serial execution time (of the $\boxed{\text{best}}$ sequential algorithm)

③ Efficiency = $Speedup_p / p$ → Fraction of time for which processing element is usefully employed (18)

④ Resource usage cost = $C = P * T_p$
- load imbalance — bad — idle time is charged
- find the sweetspot — no more scaling

metrics

Relationship between efficiency and cost = ~~efficiency~~ $\frac{T_s * \cancel{T_p}}{\cancel{T_p} * \text{efficiency}}$

$$C = \frac{T_s}{\text{efficiency}} \qquad C \propto \frac{1}{\text{efficiency}}$$

Scalability — speedup $\propto$ problem size } if this is seen ~~at~~ with a high correlation

example — summing — n/p — divide and add
— $2\log(p)$ — merge p sums to a global sum

speedup = n/T_p

Isoefficiency — to consider efficiency in terms of problem size (W), number of processing elements (p), overhead function (T)

$$E = \frac{1}{1 + T(W,p) / W} \Rightarrow W = \frac{E * T(W,p)}{1-E}$$

'k' constant

Here W(p) is isoefficiency f^n

increase problem size by W(p) to get same efficiency

observation { per message cost — if significantly larger than — per byte cost
- send fewer messages
- combine messages

α — per message cost | β — per byte cost

Computation - commn ratio = $\frac{\# \text{computation}}{\# \text{communication}}$

Little's law → concurrency = latency * bandwidth

- concurrency →
  - bytes expressed to memory subsystem
  - concurrent memory operations
- latency →
  - memory latency
  - functional unit latency
- bandwidth →
  - conventional memory bandwidth
  - # FPU

Optimizations
- memory → as superscalar processors use stream prefetchers $\xrightarrow{\text{expressing}}$ memory pattern in a streaming fashion helps
- FPU → unroll/jam the code — for independent FP operations

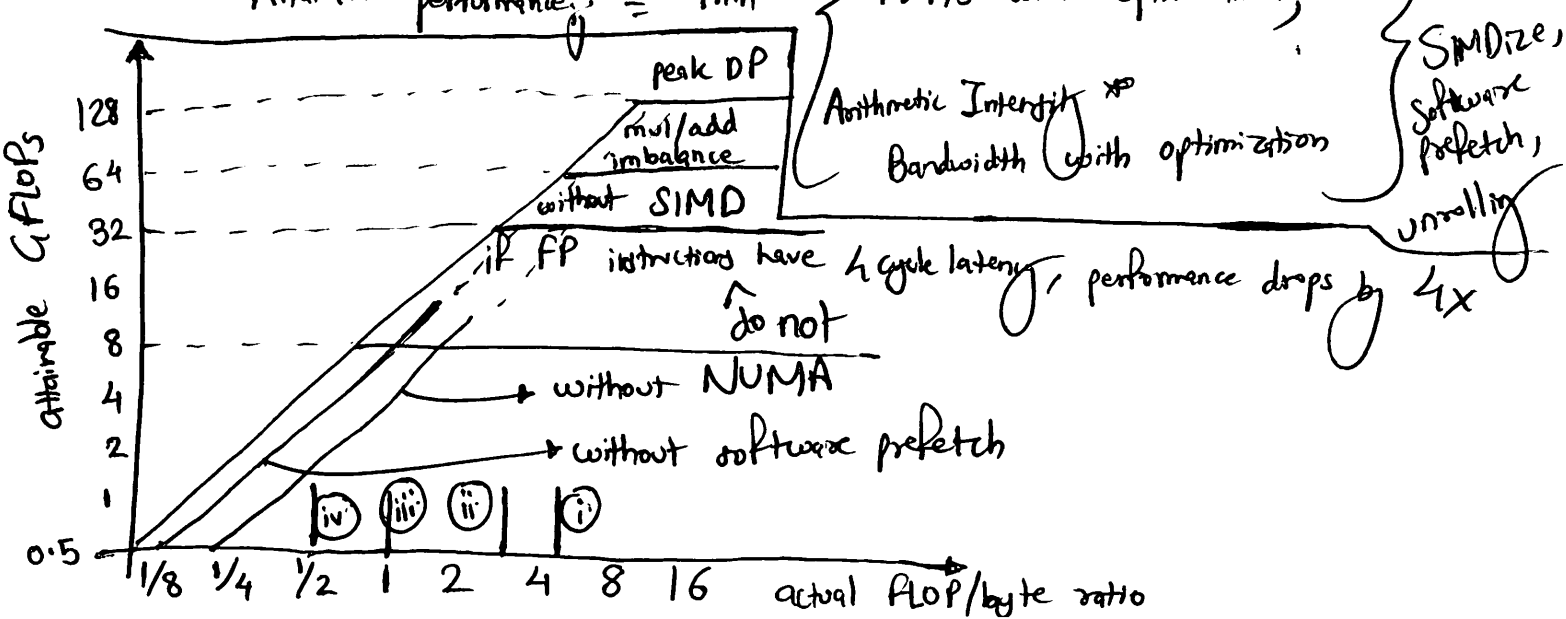
3 classes of locality
- → temporal — transform loop or algorithm to maximize reuse — reuse data multiple times
- → spatial — transform data structs → SOA (structure of arrays) — as we want every word in a line to be used
- → sequential — for stream prefetchers unroll/jam the loops

True arithmetic intensity = Total FLOPs / Total DRAM bytes

Overlap communication — time = max (time to transfer data, time for FLOPs)

Roofline model — synthesizes computation, communication, and locality

Attainable performance = min { FLOP/s with optimization, Arithmetic Intensity * Bandwidth with optimization } — optimization can be SIMDize, software prefetch, unrolling



also use vector "neigh" — "s.neigh" - vertices adjacent to all vertices in S

(20)

mul/add imbalance — opterons have dedicated multipliers and adders — gets halved

no SIMD — halved

(i) compulsory miss traffic
(ii) write allocation traffic
(iii) capacity miss traffic
(iv) conflict miss traffic

$$AI = \frac{FLOPS}{\text{conflict} + \text{capacity} + \text{allocations} + \text{compulsory}}$$

What can lower flop/byte ratios for cache behavior

- conflict, capacity, compulsory misses
- write allocated cache line getting flushed
- Array of structs layout

observation for in-core performance: as instruction mix shifts away from floating point, finite issue bandwidth affects limits on in-core performance

Optimization Categories

Maximizing in-core performance

- exploit in-core parallelism (ILP, DLP, etc.)
- good enough floating-point balance

reorder

unroll and jam

eliminate branches

explicit SIMD

Maximizing memory bandwidth

- Exploit NUMA
- Hide memory latency
- Satisfy little's law

unit-stride streams

memory affinity

SW prefetch

DMA lists

TLB blocking

Minimizing Memory traffic

- Eliminate compulsory, capacity, conflict misses and write allocation traffic

cache blocking

array padding

Streaming stores

compress data