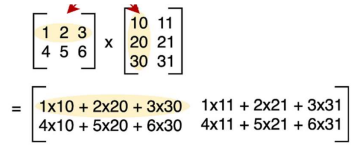

Primitives

Processing Unit
Vectorized Instructions → CPP compiler auto-vector

Decomposition

Divide the problem into tasks

- Granularity → tasks size → Static; round-robin; block
- Mapping → assignment of processes/thread to physical core/processor
- Dependencies → impose execution order constraints

Concurrency VS Parallel

Concurrency → overlapping, but might still be 1 PU executing tasks in turn
Paralllel → Multiple tasks at the same time, simultaneous

Performance

Execution time → wall clock time taken by the program to complete problem → ie n tasks
Throughput → number of tasks done in limited time

Total time = (instructionTime+MemoryTime)*number of that instruction

- Probability of cache hit/miss involved too

Measurements

MIPS; (G)FLOPS → Instructions; Floating Point Operations
Cache hit/miss rate
CPU Utilization/waiting time

SPEEDUP

Assume same algo but with parallel → some seq algo can be faster but cannot be made parallel

Scalability

Constraints: granularity of problem depends on the hardware we run on too
small problem → parallel overhead may dominate runtime
large problem → may not fit in memory of small machine → thrashing

Amdahl's law (fixed workload)

Seq cannot be sped up, par can be sped up N times on N cores
Speedup tends to increase with larger problem size + more cores → superlinear!

Gustafson's law (time constraint)

Seq time is constant, how many cores needed to make problem of size N fit within time T

Datacentre vs cloud → SOC cluster is DC because we "own" the hardware; cloud is managed service kinda deal

Processes and Threads

Process is an "independent" program in execution

- Identified by PID
- Comprises → Code, global data (os resources, files, network), stack+heap, register values

Independent address space → Inter-Process Communication is explicit
Context Switch between processes → store and restore state is ocerhead
Fork() → create a clone of process with different PID, return 0 in original, return PID in new Process

- Linux copy on write → less overhead
- Process creation still quite costly
- Process Communication also quite costly → through OS
- Windows new process is a spawn → recall URA multiprocessing stuff

Process Interactions

Exceptions → error state when executing machine instruction
Sync → occurs due to program, run exception handler
Interrupt → external event, usually IO related
Async → independently of program , interrupt handler

Threads

Independent control flows
Share address space of parent process
Same code, data, heap, files; different registers and stack
Global variables and heap objects are accessible by all threads of *same* process

User-level vs kernel threads

User level → managed by program. Switching fast, but OS cannot map threads in same process to different cores
Kernel Thread →OS can allocate them to different cores
1-to-1 mapping, but many-to-many mapping too → thread group shares some subset of all cores

Synchronization

Control interleaving to ensure correctness → scheduling is not under program control
Race condition → access shared resource with no sync and at least 1 modifies it → ie multiple possible outcomes
Control Access → Critical section

Locks

Acquire→ enter CS ; Release → exit CS

Semaphore

Stores a count of how many more can enter the CS
Wait → decrement, block until semaphore freed; Signal → Increment, allow another thread to enter
Binary and counting semaphores

Barrier

Wait for all/n threads to hit a point, then all go together

Deadlock

All waiting, no progress

1) Mutual Exclusion → at least 1 resource held in non-sharable mode
2) Hold and wait → one process holding a resource and waiting for another resource
3) No pre-emption → CS cannot be aborted externally ; ie cannot force release resource
4) Circular wait → a wait for b; b waiting on a

Starvation

Process prevented from making process due to others always holding resource

Eg always use largest PID to run first, small PID will starve

Livelock

Movement to avoid deadlock but no progress

Classical Problems

Producer-Consumer

Reader-Writer

Dining Philosopher

Hardware

Test_and_set(ref) → return old value, then sets val to true
While(test_and_set(ref)){} → busy wait until something sets ref to false

Pipelining → instructions can overlap
SuperScalar → duplicate pipelines, multiple instructions
- Hardware identifies instructions which can run simultaneously, some reordering possible

Multiprocessor → each needs its own context; register, fetch, exec etc

Flynn's Taxonomy

1) SISD → trivial
2) SIMD →Each instruction works on multiple data; eg. Vectorized AVX op or GPGPU
3) MISD → multiple instruction streams, working on same data
4) MIMD → Each PU fetch instruction, then apply to different data → as long as there's an if else, probably this option
5) SPMD → MIMD where code is same, but control flows chosen might be different (same Program

Designs

Hierarchical → different (shared) cache levels and cores at leaf of tree

Pipelined → often used in stream processing, eg router/graphics

Network → cores and memory elements connected by a network of interconnects

Memory/IO Bandwidth is usually the bottleneck

Esp with shared memory/cache loading problems

Unified Memory Access → time to memory is consistent across cores

NUMA → time to memory can vary depending on core and memory which is accessed

Shared memory

+ no need to partition data or code → dev time
+ no need to move data
- synchronization is required
- lack of scalability due to contention → memory bandwidth limit

Memory

Contention → one core having to wait on another for resource

Coherence → Same location

→ issue: write cannot be seen by other processors with older cached values
- All processing units must agree on order of reads and writes to the **same memory location**

1) Program Order
    a. Same processor read after write should get new value
2) Write Propagation
    a. P1 write to X → no further writes → P2 should read the value written by P1
    b. Ie writes become visible to all other processing units <u>eventually</u>
3) Transaction serialization
    a. Write V1 to X → Write V2 to X → cannot read X as V2, then as V1
    b. Ie all writes are seen in same order by all processing units

Can be maintained by hardware or software
Snooping based → cache snoops on bus, updates status of cache lines
Directory based → sharing status kept in centralized location

Implications

1) Overhead in shared address space → CC appears as increase latency in memory access
    a. Lowers cache hit rate
2) Cache ping pong
    a. When multiple PU read and modify same global variable
    b. Gets invalidated in all PU and need to get new value from main memory
3) False sharing
    a. 2 PU write to different addresses but same cache line → eg diff elements of struct
    b. Same effect as ping pong, but hidden

Consistency → different locations

→ constraints on the order in which memory operations by one thread become visible to other threads for **different** memory locations
Relaxing Consistency → hide write latencies by reordering operations
4 types of memory reordering ; x and y are diff variables in same thread

1) W → R : write to X must commit before read from Y
2) R → R : read from X must commit before read from Y
3) R → W : read from X must commit before write to Y
4) W → W : write to X must commit before write to Y

Sequentially Consistent

Interleaving can, no reordering

Total Store Ordering

Relax R→W

Can read B before prior write to A is seen by all Pus → A = 1;print B; can be reordered to print B; A=1;

Reads by other PU cannot return new value of A until new A is observed by all PU (write atomicity)

Processor Consistency

Return value of any write (even from another PU) before the write is observed by all processing units

Serialization and propagation preserved, atomicity not

Can be read by some before others → If P1 reads A=1 to exit busy wait, no guarantee that P2 executing afterward will read A=1 too

Partial Store Ordering

Relax W→R and W→W

Writes can be reordered if different target

// A=1;Flag=1; can be reordered to Flag=1;A=1;

Patterns

Fork Join

- Parent creates child task, children are independent, children might join parent at some future time

Parbegin-Parend

- "narrower" version of Fork Join where all fork and all join at the same time
- Specify a series of statements to be executed in parallel, threads usually execute the same code

SIMD/SPMD

- Same program on different cores, different data
- No explicit synchronisation
- Eg GPGPU

Master-Worker

- A3 basically
- 1 master allocating work to multiple workers

Task Pool

- Fixed number of threads
- Workers take jobs from common pool (! Not allocated by master!) and execute then, can generate more tasks into main pool

Producer-Consumer

- Producers feed into a buffer, consumers read from it

Pipelining

- Each worker/node is a step, they pass the data from node to node

Data vs Task

Data parallel → same OP to different data

- Elements of array/matrix being independently processed

Task parallel → Each task is different, and pass results back and forth

- Assignment 3 is task because of children

Foster's Design Methodology

1) Partition → divide the problem into smallest possible tasks
2) Communication → provide data for each task → but minimise bandwidth required
3) Agglomeration → combine tasks to decrease communication and cost
4) Mapping → Map tasks to processors

Models2

Data distribution

1D distribution

Blockwise → chunk then give contiguous sections

Cyclic → round robin

Block Cyclic → make blocks of predefined size, cyclic based on these blocks

2D

Block Cyclic → split by col/row and treat other dimension as 1d array of arrays

Checkerboard → divide grid in both dimensions, then distribute by any of the above

- Checkerboard has better locality in both dimensions, and defined size
- Block cyclic runs risk of ending up with just 1 col/row if data too big, or not fitting in memory

Message Passing

Send receive

Blocking non-blocking

Buffered unbuffered

Synchronous vs asynchronous is Global view

Sync→ communication does not complete until both sides have started an operation

Asynchronous → sender can execute communication without coordination with receiver

Blocking non-blocking is Local view

Blocking → I must wait for the buffer to be safe before the function returns

Non-blocking → procedure may return before op is completed; I must check if I want to use the resources

CUDA/GPGPU

Device: GPU; Host: CPU; Kernel: function that runs
Kernel is the "main" CUDA function executed by array of threads
Each thread has an id unique within block → blockID.xyz, threadID.xyz
mykernel<<<blocks, threads, shared_mem, stream>>>(args);
kernel<<<dim2 grid, dim3 block, int smem, int stream>>>(...)
Kernel is organised as a *GRID* of *BLOCKS* containing *threads*

- Executed in a warp, which is a group of 32 threads

GRID does not share memory → threads outside block cannot cooperate

BLOCK

Threads in block share memory, have atomic ops, barrier synchronization

Hardware can schedule thread blocks onto any processor at any time

Threads in same block (warp) execute in lockstep → SIMD

Multiple blocks can exist on one SM

- Register file and shared memory partitioned among all resident thread blocks

Single Instruction Multiple Thread

Threads in a warp all start at same program address, but have own PC and register state

CUDA memory

Must be explicitly transferred

Global Memory → cached

Shared Memory → uncached

Local Memory → automatic array variables allocated by compiler

Constant Memory → uniform access read only

Texture memory → spatially coherent random access read only

Coalesced access to global memory → if threads in warp want data close to each other (within same 32 byte transaction), then it is transferred as 1 transaction

Shared Memory → divided into banks

- Addresses from different banks can be accessed simultaneously
- Bank conflict → different threads want data from same bank

Optimizing CUDA

Minimize data transfer between host and device

Coalesce global memory access

Minimize global memory access by using shared memory

Minimize bank conflict in shared access

Minimize control flow divergence in block

OpenMP

Pseudocode

`Reduction` keyword means reduction is done in parallel → probably binary tree style

- Should be an associative a+(b+c)==(a+b)+c and commutative op a+b == b+a

MPI

Overhead of correctness

- Idling → non-buffered
- Buffer management → buffered

Side effect

- Safe and easier → all blocking
- Hide comm overhead → non-blocking

Rank is identifier

Sender sending 2 messages to same receiver → receiver will get them in the same order

Deadlocks can be avoided if blocking sends goes into buffer → eg both send then receive

Virtual Topology

Good for modelling different relationships, eg ring/line/grid

Interconnections

Physical connections between different nodes in a distributed network

The choice type of algo to use often depends on how things are connected

- Less communication overhead

Topologies

Direct vs indirect

Hypercube → connected to everything where numbering differs by 1 bit

Cube Connected Cycle → 3 connections per node

- From a K dimension hypercube, replace each node with a cycle of K nodes
- Then each node has 2 connections to its ring + 1 to the outside

Summary of Metrics

network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity $ec(G)$	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hypercube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k-1+\lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2\log \frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Indirect Connections

Use switches eg ethernet protocol

Bus network → old ethernet/wifi kinda protocol

Crossbar network → Processors on left, Memory on bottom, grid of switches that can control straight or swap

Adaptive routing might take into account network status/congestion/dead nodes etc

Deterministic always uses same path

Omega Network

Log n stages, n/2 switches per stage

Construction → cyclic left shift the binary label of the switch, connect to the shifted and the flipped LSB of shifted

XOR tag routing → let T = source XOR dest, then at each stagek, straight if T[k]==0 else crossove

Energy Effiiency

Heterogenous cores → ARM/Phones with big and little cores
- Decrease idle power consumption

Power Use Efficiency (PUE)
- Total energy used / amount needed for the processors

OpenMP

#pragma omp parallel for
// for loop
Nested work sharing → not allowed, undefined behaviour

Schedule (static, chunksize) → less overhead
Dynamic → get fixed chunk size when idle
Barrier →
Master → define a region that only the master thread executes
Critical → define a CR that only 1 thread can enter
Atomic → define memory location (variable) which must be updated atomically
Section → define sections which will be assigned to available thread one at a time
Reduce → collect all the individual values of this variable using the provided OP
Shared/Private variables → explicit and implicit
- Reduce should be a private variable

MPI

Send Recv
Isend Irecv
Scatter Gather → even amount
scatterv → send unequal amounts
Broadcast / MultiBroadcast → Tree-style broadcast
- Broadcast is root to all
- Multibroadcast is all to all, all receivers receive 1 data piece from every other worker in group
    - Sorted in rank order

Multi Accumulation
- Each processor provides a diff data block to each, eg elem[dest], then do sth with all they receive

Total exchange → "transpose"
Multi-accumulation w/o the reduction step
Wait/Test
Barrier
Virtual topologys

Map/Bind

Default 1 thread per slot → slot can be CPU/core/device etc
Map: which process run on which node
Bind: constrain process to specific CPU
Binding usually match mapping if its an "object-specific" mapping
Oversubscribe→ more processes than processing units → lab exercise where "idle" process caused blocking on every other live process