CS3210 Parallel Computing

AY 25/26 Sem 2 — github/omgeta

1. Introduction

Parallel Computing is the simultaneous use of multiple processing units to solve problems efficiently.

Program Parallelization Steps:

i. Decomposition: split problem into parallel tasks
   - Granularity: size of task

ii. Scheduling: assign tasks to processes/threads
   - Orchestration: imposed synchronisation and communication of tasks to satisfy dependencies

iii. Mapping: bind processes/threads to hardware processing units (e.g. CPU cores)

Processes

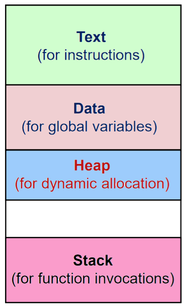
Process is an abstraction for a running program:

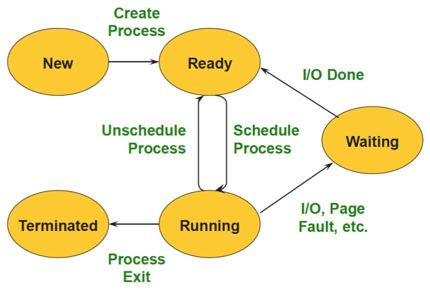
i. Process ID (PID): uniquely identifies a process

ii. Process State: indicates execution status

iii. Process Control Block: stores execution context (registers, resources, exclusive address space)

− Costly: syscall overhead, data structures must be allocated, communication goes through OS





Interprocess Communication (IPC):

i. Shared Memory (e.g. locks, semaphores)

ii. Message Passing

iii. UNIX Pipes and Signals

Threads

Threads are independent execution flows within a process:

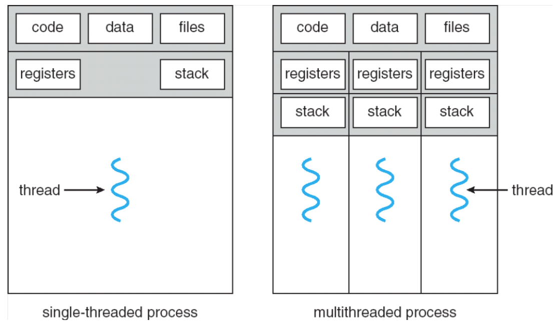
i. Shared: Resources, Address Space

ii. Private: Thread ID, Registers, "Stack" (diff. SP)

+ Efficient: cheaper creation and context switching

+ Resource Sharing: process resources can be shared

+ Concurrent: multithreading on different cores



Implementations:

i. User-Threads are managed by a userspace library:
   + Fast: context switches have low overhead
   − No Parallelism: OS cannot map threads to different execution units
   − Threads blocking will block whole process

ii. Kernel-Threads are implemented in the OS:
   + Parallel: threads can run across many CPUs
   − Slower: thread operations are system calls

Mapping Models:

i. Many-to-One:
   - All user-level threads are mapped to one process
   - User library schedules user-threads

ii. One-to-One:
   - Each user-thread mapped to one kernel-thread
   - No user library scheduler needed

iii. Many-to-Many:
   - User-threads mapped to set of kernel-threads
   - Library scheduler may move user-threads to different kernel-threads

Synchronisation

Synchronisation is required by concurrent threads to coordinate access to shared resources in critical sections.

Critical Section Properties:

i. Safety: nothing bad happens
   - Mutual Exclusion: at most 1 thread in CS

ii. Liveness: something good happens
   - Progress: if no thread is in CS, a waiting thread should be granted access
   - Bounded Wait: a waiting thread requesting to enter the CS, will eventually enter

iii. Performance: overhead of entering/exiting CS is small w.r.t work done within it

Symptoms of Incorrect Synchronization:

i. Deadlock: all threads blocked; iff all conditions met
   - Mutual Exclusion: ≥ 1 resource held exclusively
   - Hold & Wait: ≥ 1 process holding one resource while waiting for another
   - No Pre-emption: resources must be yielded
   - Circular Wait: set of processes waiting in circles

ii. Livelock: due to deadlock avoidance, no progress

iii. Starvation: a thread is unable to access CS

iv. Race Condition: bug from program outcome depending on unpredictable access order

Mechanisms:

i. Lock: primitive `acquire()`, `release()`
   - Spinlock: busy-wait; wastes cycles
   - Mutex: blocking
   - Uses hardware atomics (`TSL`) or toggle interrupts

ii. Semaphore: atomic counter ≥ 0; `wait()`, `signal()`
   - Binary/Mutex: represents single access
   - Counting/General: number of threads in CS $\leq N$
   - Can be manipulated by different thread
   - No relation to data being controlled

iii. Monitors: thread-safe high-level data structures

iv. Messages

Producer-Consumer

Processes share a bounded buffer of fixed size K, where producers add items until buffer full and consumers remove items when not empty.

Blocking Solution (init `not_full = K, not_emp = 0`):

```
// Producer
x = produce();

wait(not_full);
wait(mutex);
buffer.add(x);
signal(mutex);
signal(not_emp);
```

```
// Consumer
wait(not_emp);
wait(mutex);
x = buffer.get();
signal(mutex);
signal(not_full);

consume(item);
```

Busy Waiting Solution:

```
// Producer
while(!can_prod);
x = produce();

wait(mutex);
if (K > count) {
  buf[in] = x;
  in = (in+1)%K;
  count++;
  can_cons = 1;
} else {
  can_prod = 0;
}
signal(mutex);
```

```
// Consumer
while(!can_cons);

wait(mutex);
if (count > 0) {
  x = buf[out];
  out = (out+1)%K;
  count--;
  can_prod = 1;
} else {
  can_cons = 0;
}
signal(mutex);

consume(x);
```

Reader-Writer

Processes share a critical region, where readers can read simultaneously but writers must have exclusive access.

```
// Writer
wait(empty);
write();
signal(empty);
```

```
// Reader
wait(mutex);
readers++;
if (readers == 1)
  wait(empty);
signal(mutex);

read();

wait(mutex);
readers--;
if (readers == 0)
  signal(empty);
signal(mutex);
```

– Writer Starvation: possible if readers keep entering and `empty` is never signalled. Solutions:
  - Queue: all pass through `wait(queue)`
  - Writer-Priority: readers entering must `wait(readTry)`; but first waiting writer blocks new readers with `wait(readTry)` and does `signal(readTry)` when done

Barrier

```
int i_am_last = 0;
wait(mutex);
count++;
if (count == N) {
  i_am_last = 1;
  signal(barrier);
}
signal(mutex);

wait(barrier);
if (!i_am_last)
  signal(barrier);
```

Dining Philosophers

N philosophers around a circular table, with a single chopstick between. 2 chopsticks are needed to eat.

```
// Philosopher
void p(int i) {
  while (1) {
    // think
    take(i);
    eat();
    put(i);
  }
}

void safe_to_eat
   (int i) {
  if (state[i] ==
   HUNGRY &&
   state[LEFT] !=
   EATING &&
   state[RIGHT] !=
   EATING) {
    state[i] =
   EATING;
    signal(s[i]);
  }
}
```

```
// Take Chopstick
void take(int i) {
  wait(mutex);
  state[i] =
   HUNGRY;
  safe_to_eat(i);
  signal(mutex);
  wait(s[i]);
}

// Put Chopstick
void put(int i) {
  wait(mutex);
  state[i] =
   THINKING;

   safe_to_eat(LEFT);

   safe_to_eat(RIGHT)
  signal(mutex);
}
```

Limited Eater Solution, when at most $N-1$ philosophers eat concurrently, it's guaranteed at least one can eat:

```
void philosopher(int i) {
  while (1) {
    // think
    wait(seats);
    wait(chopstick[LEFT]);
    wait(chopstick[RIGHT]);
    eat();
    signal(chopstick[RIGHT]);
    signal(chopstick[LEFT]);
    signal(seats);
  }
}
```

2. Architecture

Levels of Parallelism:

i. Bit-Level: operate on wider words

ii. Instruction-Level:
   - Pipeline: split execution into stages; multiple instructions can occupy different stages in same cycle (if no dependency hazards)
   - Superscalar: duplicate pipelines; issue multiple independent instructions per cycle

iii. Thread-Level: execute multiple threads on the same core concurrently

iv. Processor-Level: execute multiple threads/processes in parallel on multiple cores/processors

Flynn Taxonomy of Parallel Architectures:

i. Single Instruction Single Data (SISD)

ii. Single Instruction Multiple Data (SIMD)

iii. Multiple Instruction Single Data (MISD)

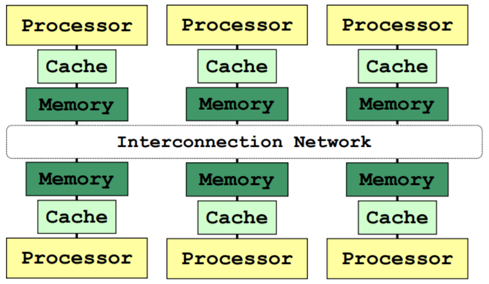
iv. Multiple Instruction Multiple Data (MIMD)

Distributed-Memory

Distributed-Memory Systems are multicomputer systems:

i. Node: indep. unit of processor and private memory

ii. Interconnection Network: supports message passing for data exchange between nodes



Shared-Memory

Shared-Memory Systems are multiprocessor systems:

i. Shared Memory Provider: maintains shared address space abstraction across processors

+ No need to partition code or data

+ Efficient communication: no need to physically move data between processors

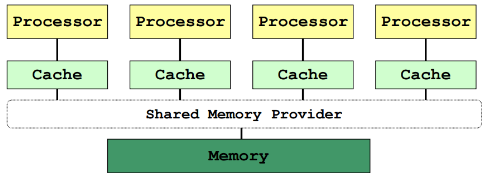
– Special synchonisation constructs required

– Limited scalability due to contention

Uniform Memory Access (UMA):

i. Same memory access latency

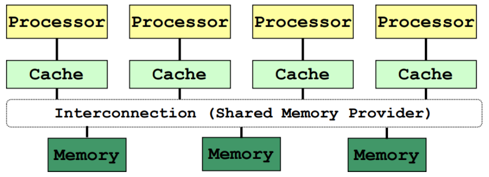
+ Suitable with few processors due to contention



Non-Uniform Memory Access (NUMA):

i. Different memory access latency for every processor (e.g. local memory faster than remote)

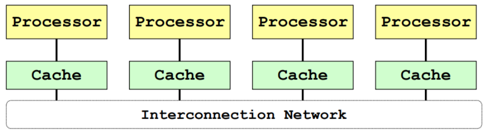
ii. Cache Coherent NUMA (ccNUMA): each node has coherent cache to reduce contention



Cache Only Memory Access (COMA):

i. Memory blocks behave as caches

ii. Data migrates dynamically and continuously according to cache coherence scheme



Cache Coherence

Caches reduce memory access latency and provide high bandwidth data transfer to CPU:

i. Cache Size: larger cache reduces misses but increases access time (addressing complexity)

ii. Block Size: unit of transfer; larger blocks improve spatial locality, but longer transfer time

Write Policy:

i. Write-through: write to both cache and memory
   + Always get newest value of memory block
   – Slower due to many memory accesses; mitigate with write buffer

ii. Write-back: write to cache only and set dirty bit; write to memory on cache block eviction
   + Fewer write operations to memory
   – Memory may contain invalid entries

Cache Coherence ensures all processors have consistent view of memory through their local cache. Properties:

i. Program Order: if P writes to X, then n.f.w to X, P should read same value from X

ii. Write Propagation: if P_1 writes to X, n.f.w to X, P_2 should read same value from X

iii. Write Serialization: if $V_1 \to X$, then $V_2 \to X$, all P_i should never read V_2 then V_1 from X

Implementations:

i. Software-based: use page-fault to propagate writes

ii. Snooping-based: no centralized directory; caches snoop a shared bus and react to transactions

iii. Directory-based: centralized directory records sharing status; common in NUMA

Implications:

i. "Increase" memory latency and lower cache hit rate

ii. Cache Ping-Pong: multiple processors repeatedly read/modify same shared variable

iii. False Sharing: processors write to different addresses which map to same cache line

Memory Consistency

Memory Consistency constrains the in which order of memory operations by a thread become visible to other threads across different memory locations.

i. Programmer: helps reason about correctness and program behaviour
ii. System/Compiler: decide what memory reordering is allowed (to hide write latencies)

Constraints:

i. $W \to R$: write to X completes before read from Y
ii. $R \to R$: read from X completes before read from Y
iii. $R \to W$: read from X completes before write to Y
iv. $W \to W$: write to X completes before write to Y

Sequential Consistency Model (SC):

i. Processors issue mem. operations in program order
ii. Result is as if all operations were interleaved in one sequential order seen consistenly by all processors

+ Intuitive
+ Maintains all constraints
− Can result in loss of performance

Relaxed Consistency Model:

i. Relax ordering if data dependencies allow (i.e. memory operations to same memory location):
   - $R \to W$: anti-dependence (WAR)
   - $W \to W$: output dependence (WAW)
   - $W \to R$: flow dependence (RAW)
ii. Allow overriding mechanism for programmers
iii. Write-to-Read Program Order:
   - Total Store Ordering (TSO): return value written earlier without waiting for serialization
   - Processor Consistency (PC): return value of any write before propagation or serializion
iv. Write-to-Write Program Order:
   - Partial Store Ordering (PSO): later write can propagate/serialize before earlier write

+ Hide latencies by overlapping indep. operations

Interconnection Networks

Interconnection forms the backbone of communication between processors, memories/caches, and I/O devices.

Topology

Topology is the geometrical shape of the connection.

Direct Interconnections (or Static, P2P) have endpoints connected directly, usually of same type (e.g. core–core). Metrics:

i. Diameter, $\delta(G)$: maximum shortest-path distance between any pair of nodes
   - Small diameter $\Rightarrow$ shorter message transmission
ii. Degree, $g(v)$: no. of nodes adjacent to node v; $g(G)$ is the maximum node degree in network G
   - Small degree $\Rightarrow$ lower node hardware overhead.
iii. Bisection Width, $B(G)$: minimum no. of edges removed to split the network into two equal halves
   - Bisection bandwidth $BW(G)$: total bandwidth between the two bisections
   - Measures capacity under simult. transmission
iv. Node Connectivity, $nc(G)$: minimum no. of nodes whose failure disconnects the network
   - Determines robustness of the network
v. Edge Connectivity, $ec(G)$: minimum no. of edges whose failure disconnects the network
   - Determines number of independent paths between any pair of nodes

	n	$g(G)$	$\delta(G)$	$ec(G)$	$B(G)$
Complete Graph	n	$n-1$	1	$n-1$	$(\frac{n}{2})^2$
Linear Array	n	2	$n-1$	1	1
Ring	n	2	$\lfloor\frac{n}{2}\rfloor$	2	2
d-dim Mesh	r^d	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dim Torus	r^d	$2d$	$d\lfloor\frac{\sqrt[d]{n}}{2}\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dim Hypercube	2^k	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dim CCC	$k2^k$	3	$2k-1+\lfloor\frac{k}{2}\rfloor$	3	$\frac{n}{2k}$
Complete BT	2^k-1	3	$2\log\frac{n+1}{2}$	1	1
k-ary d-cube	k^d	$2d$	$d\lfloor\frac{k}{2}\rfloor$	$2d$	$2k^{d-1}$

Indirect Interconnection (or Dynamic) have endpoints interconnect through switches configured dynamically:

+ Reduce hardware costs by sharing switches & links

Metrics:

i. Cost: number of switches and links
ii. Concurrent connections

Types:

i. Bus: set of wires carry data from sender to receiver
   - Only a pair of devices can communicate at a time
   - Bus arbiter used for coordination
   - Typically used for small number of processors
ii. Crossbar: switch matrix (of $n \times m$ switches) from n inputs to m outputs
   - Switch states: straight or direction change
   - Hardware costly $\to$ small number of processors
iii. Multistage Switching: intermediate switches with connecting wires between neighbouring stages
   - Goal: obtain small distance for any pair of input and output devices
iv. Omega: unique path from each input to each output $(n \times n)$
   - Construction: switch $(\alpha, i) \to (\beta, i+1)$ where β is cyclic-left-shift of α, and cyclic-left-shift + inversion of LSB
   - $\log n$ stages of $\frac{n}{2}$ switches per stage
v. Butterfly:
   - Construction: switch $(\alpha, i) \to (\alpha, i+1), (\alpha', i+1)$ where α, α' differ in the $(i+1)^{th}$ bit from the left
vi. Baseline:
   - Construction: switch $(\alpha, i) \to (\beta, i+1)$ where β is cyclic-right-shift of last $(k-i)$ bits of α, and inversion of LSB + same cyclic-right-shift

Routing

Routing determines path(s) from source to destination within a given topology. Algorithm Classification:

i. Path Length:
   - Minimal: shortest-path always chosen
   - Non-minimal: shortest-path not necessary

ii. Adaptivity:
   - Deterministic: always use same path for same pair of source and destination nodes
   - Adaptive: take into account network status (e.g. avoid congested path, avoid dead nodes)

Deterministic Algorithms:

i. XY Routing for 2D Mesh:
   - Move in X direction until $X_{src} == X_{dst}$
   - Move in Y direction until $Y_{src} == Y_{dst}$

ii. E-Cube Routing for Hypercube:
   - Bit difference in source and target node address (hamming distance) = number of hops
   - From MSB→LSB (or vice versa), find first different bit and go to neighbour node with bit corrected; at most n hops

iii. XOR-Tag Routing for Omega Network:
   - Let T = Source ID $\oplus$ Destination ID
   - At stage k, go straight if bit $k = 0$, else crossover

Additional

More Questions:

i. Switching: how to transfer messages along a path
ii. Flow Control: how to handle concurrent messaging

Current Trends

Ethernet: supports P2P and broadcast; common topologies include bus, and star with a logical bus.

InfiniBand: supports P2P and multicast; common topologies include fat-tree and torus.

3. Programming Models

Types of Parallelism:

i. Data Parallelism: same op. on different data elements (loop parallelism; SIMD/SPMD)

ii. Task Parallelism: independent tasks executed concurrently (statements, loops, function calls)
   - Task Dependency Graph: DAG representing control dependency between tasks
     - Critical Path Length: slowest completion
     - Deg. Concurrency: $\dfrac{\text{Total Work}}{\text{Critical Path Length}}$

Representation of Parallelism:

i. Implicit Parallelism: Haskell
ii. Implicit Scheduling: OpenMP
iii. Explicit Communication & Sync: MPI, Pthreads

Models of Coordination:

i. Shared Address Space:
   - Communicate via shared variables; mutual exclusion via locks
   - Matches shared-memory (UMA/NUMA); requires HW support; costly to scale

ii. Data Parallel:
   - Map a function onto a large collection of data (historic: SIMD/vector; modern: CUDA)
   - Rigid structure enables parallel scheduling; ideally no communication among invocations

iii. Message Passing:
   - Private address spaces; explicit send/receive
   - Matches distributed memory
   - Point-to-point vs Global communication
   - Blocking vs Non-Blocking
   - Buffered vs Non-Buffered
   - Synchronous vs Asynchronous

Foster's Design Methodology

i. Partitioning: problem into pieces/data
   - $\geq 10\times$ primitive tasks than cores, of similar size
   - Minimize redundant computations and data
   - Number of tasks increases with problem size

ii. Communication: pass needed data among tasks
   - Balance communication among tasks
   - Perform communication in parallel
   - Overlap computation with communication
   - Communicate with small number of neighbours

iii. Agglomeration: combine tasks into larger tasks
   - Number of tasks $\geq$ number of cores
   - Reduce cost of task creation + communication
   - Maintain scalability, simplify code

iv. Mapping: tasks to execution units
   - Maximize processor utilization; minimize IPC
   - Optimal mapping is NP-hard ⇒ heuristics
   - Consider static vs dynamic allocation;
     if static, tasks:cores $\geq$ 10:1;
     if dynamic, allocator must not bottleneck

Parallel Programming Patterns

Patterns:

i. Fork–Join: `fork` child tasks, then `join`
ii. Parbegin–Parend: mark statements to execute in parallel, then sync at end
iii. SIMD: single instruction stream over multiple data elements (synchronous)
iv. SPMD: same program on multiple threads; different data via IDs (asynchronous)
v. Master–Worker: master distributes tasks to workers
vi. Client–Server (MPMD): request-response model
vii. Task Pools: shared pool of tasks; workers dynamically dequeue tasks
viii. Producer–Consumer: produce tasks into buffer for consumers to retrieve; needs sync
ix. Pipelining: split compute into stages for data stream; throughput improves via stage-overlap

OpenMP

Shared-Memory Programming Model:

i. Threads communicate via shared variables; may also have private variables
ii. Avoid race conditions using mutual exclusion

Parallel For (Data Parallel)

```
// Parallel outer loop; shared data +
   private indices
#pragma omp parallel for num_threads(8)
shared(a,b,result) private(i,j,k)
for (i = 0; i < size; i++)
  for (j = 0; j < size; j++)
    for (k = 0; k < size; k++)
      result[i][j] += a[i][k] * b[k][j];
```

Mutex with OpenMP Lock

```
int count = 0;
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel
{
  omp_set_lock(&lock);
  count = count + 1;
  omp_unset_lock(&lock);
}
```

NVIDIA CUDA

CUDA Programming Model:

i. Heterogeneous: Host (CPU) + Device (GPU)
ii. Kernel: function that runs on the device
iii. Thread: local registers and memory; `threadIdx`
iv. Warp: execution unit of 32 threads scheduled together (SIMT); one instruction at a time
v. SM (Streaming Multiprocessor): schedules warps with resources (register file, shared memory); thread blocks execute concurrently on an SM

Function Qualifiers:

Qualifier	Scope	Usage
`__host__`	CPU	local to host
`__device__`	GPU	local to device
`__global__`	GPU	launch from host

Variable Qualifiers:

Qualifier	R/W	Cache	Scope	Lifetime
`__device__`	R/W	No	grid	static
`__constant__`	R	Yes	grid	static
`__shared__`	R/W	No	block	block
unqualified	R/W	–	thread	thread
`__managed__`	R/W	–	global	static

Optimizations:

i. Reduce transfers: minimize host↔device copies; batch small copies; use pinned memory; overlap copy/compute with `cudeMemcpyAsync` + streams
ii. Global memory: coalesce accesses; minimize global loads/stores; use `__shared__`; avoid bank conflicts
iii. Occupancy: many warps to hide latency; threads/block multiple of 32; balance registers/shared-mem so ≥ 1 block/SM
iv. Control flow: minimize warp divergence (avoid thread-ID dependent branches)
v. Compute: prefer high-throughput ops; single precision faster; avoid int div/mod (use shifts/bitwise)
vi. Avoid multi-context: do not create multiple CUDA contexts per GPU within one application

Matrix Addition (Device)

```
__global__
void addMatrixG(float *a,float *b,float *c,int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j * N;
  if (i < N && j < N)
    c[index] = a[index] + b[index];
}
```

Matrix Multiplication (Device)

```
__global__
void addMatrixG(float *a,float *b,float *c,int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j * N;
  if (i < N && j < N)
    c[index] = a[index] + b[index];
}
```

Host Skeleton

```
void main(){
  // Assume host arrays: h_A, h_B, h_C; device
    pointers: d_A, d_B, d_C
  size_t bytes = (size_t)N * sizeof(float);
  cudaMalloc((void**)&d_A, bytes);
  cudaMalloc((void**)&d_B, bytes);
  cudaMalloc((void**)&d_C, bytes);

  // Copy inputs Host -> Device
  cudaMemcpy(d_A, h_A, bytes,
    cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, bytes,
    cudaMemcpyHostToDevice);

  // Configure grid/block and launch kernel
  dim3 dimBlk(BLOCK_X, BLOCK_Y);    // e.g.
    (16,16)
  dim3 dimGrd((NX + BLOCK_X - 1)/BLOCK_X,
              (NY + BLOCK_Y - 1)/BLOCK_Y);
  kernel<<<dimGrd, dimBlk>>>(d_A, d_B, d_C, ...);

  // Copy result Device -> Host
  cudaMemcpy(h_C, d_C, bytes,
    cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

MPI

Message Passing Model:

i. Distributed memory
ii. Explicitly represent parallelism with send/receive

MPI Program Overview:

i. Initialize communications
ii. Communicate to coordinate computation
iii. Exit from message-passing system

Communication Semantics

Protocol Choice:

i. Buffered: sender returns after copying data into a communication buffer (less idling; buffer overhead)
ii. Non-buffered: sender blocks until matching receive is encountered (idling + deadlock risks if mismatch)

Local View:

i. Blocking: return ⇒ safe to reuse resources/buffers
ii. Non-blocking: may return before completion; programmer must ensure completion (poll/check)

Global View:

i. Synchronous: operation does not complete before both sides have started
ii. Asynchronous: sender can proceed without coordination with receiver

Operations:

i. Sync + Blocking: `MPI_Ssend`
ii. Async + Blocking: `MPI_Send` / `MPI_Recv`
iii. Sync + Non-blocking: `MPI_Issend`
iv. Async + Non-blocking: `MPI_Isend` / `MPI_Irecv`

* Note: blocking and non-blocking ops can be mixed

Library Functions

Initi, Finalize, Abort Calls:

i. `int MPI_Init(int* argc, char** argv[])`
ii. `int MPI_Finalize(void)`
iii. `int MPI_Abort(MPI_Comm comm, int errorCode)`

Point-to-Point Messaging Calls:

i. `int MPI_Send(void* buf, int count, MPI_Datatype dt, int dst, int tag, MPI_Comm c)`
ii. `int MPI_Recv(void* buf, int count, MPI_Datatype dt, int src, int tag, MPI_Comm c, MPI_Status *status)`

Message Format:

i. Data: start-buffer + count + datatype
ii. Envelope: destination/source (rank) + tag + communicator
iii. Receive buffer must be $\geq$ message length
iv. Wildcards: `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
v. `MPI_Status`: `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`

Ordering:

i. 1 sender, 1 receiver: messages delivered in order
ii. > 2 processes: message delivery order undefined

Deadlocks

Deadlock Types:

i. Message Order: both sides wait on receives
ii. Buffer: no/small buffers ⇒ sends cannot complete

Secure MPI program: correctness does not depend on assumptions about MPI runtime properties.

- Specify send/receive order
  (e.g. even: send→recv, odd: recv→send)

Collective Communication

Process Groups are ordered sets of processes:

i. Unique rank identifier
ii. Processes may be in multiple groups
iii. Communicator: communication domain for a group
   - Intra-communicator: collectives in a group; default `MPI_COMM_WORLD`
   - Inter-communicator: P2P between two groups

Virtual Topology is a Cartesian-style communicator:

i. `MPI_Cart_create`, `MPI_Cart_get`, `MPI_Cartdim_get`, `MPI_Cart_coords`, `MPI_Cart_rank`, `MPI_Cart_shift`

Collectives involve all processes in a communicator:

i. Total Exchange: `MPI_Alltoall`
ii. Multi-broadcast: `MPI_Allgather`
iii. Multi-accumulation: `MPI_Reduce_scatter`
iv. Scatter/Gather: `MPI_Scatter` / `MPI_Gather`
v. Single Broadcast: `MPI_Broadcast`
vi. Single Accumulation: `MPI_Reduce`
vii. Barrier: `MPI_Barrier`

Hello World (Master Sends)

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
tag = 100;
if (rank == 0) {
  strcpy(message, "Hello World 2");
  for (i=1; i<size; i++)
    MPI_Send(message, 14, MPI_CHAR, i, tag,
    MPI_COMM_WORLD);
} else {
  MPI_Recv(message, 14, MPI_CHAR, 0, tag,
    MPI_COMM_WORLD, &status);
}
printf("node %d : %.13s\n", rank, message);
MPI_Finalize();
```

4. Performance

Performance is measured by primary goals:

i. Response Time: wall-clock duration of an execution
   - User CPU Time: time spent on program
   - System CPU Time: time spent on OS routines
   - Waiting Time: time spent waiting for I/O and execution of other programs (timesharing)

ii. Throughput: average work completed per unit time

Sequential Programs

User CPU Time:

$$T_{user} = N_{cycle} \times T_{cycle}$$

i. $T_{cycle} = \dfrac{1}{\text{clock rate}}$

ii. $N_{cycle} = \sum n_i \times CPI_i \;\approx\; N_{instr} \times CPI$

User CPU Time (Memory Access; One-level Cache):

$$T_{user} = \left(N_{cycle} + N_{mm\_cycle}\right) \times T_{cycle}$$

i. $N_{mm\_cycle} = N_{read\_cycle} + N_{write\_cycle}$

ii. $N_{read\_cycle} = N_{reads} \times R_{read\_miss} \times N_{miss\_cycles}$

Average Memory Access Time (Two-level Cache):

$$T_{read} = T_{hit} + R_{miss} \times T_{miss}$$

i. $T_{read} = T_{hit}^{L1} + R_{miss}^{L1} \times T_{miss}^{L1}$

ii. $T_{miss}^{L1} = T_{hit}^{L2} + R_{miss}^{L2} \times T_{miss}^{L2}$

iii. Global miss rate: $R_{miss}^{L1} \times R_{miss}^{L2}$

Throughput:

i. $MIPS = \dfrac{N_{instr}}{T_{user} \times 10^6} = \dfrac{clock\_freq}{CPI \times 10^6}$

ii. $MFLOPS = \dfrac{N_{fl\_ops}}{T_{user} \times 10^6}$

Parallel Programs

Parallel Execution Time on p processors, T_p consists:

i. Time for executing local computations

ii. Time for exchange of data between processors

iii. Time for synchronization between processors

iv. Waiting time

Cost:

$$C_p = p \times T_p$$

i. Measures total work performed by all processors

ii. Cost-optimal: parallel program executes same total number of operations as fastest sequential program

Speedup:

$$S_p = \frac{T_*}{T_p}$$

i. Measures the benefit of parallelism compared to best sequential algorithm time T_*

ii. Theoretically, $S_p \leq p$ always holds

iii. In practice, $S_p > p$ (superlinear speedup) can occur (e.g. problem working set "fits" in the cache)

Efficiency:

$$E_p = \frac{T_*}{C_p} = \frac{S_p}{p} = \frac{T_*}{p \times T_p}$$

i. Measures actual degree of speedup performance achieved compared to maximum

ii. Ideal speedup $S_p = p \;\Rightarrow\; E_p = 1$

Scalability

Scalability is the interaction between problem size and parallel machine size:

i. Impacts: load balancing, overhead, arithmetic intensity, locality (application dependent)

ii. Fixed problem size:
   - Small n: overheads dominate $\Rightarrow$ suitable for small machine, poor on large machine
   - Large n: working set may not fit on small machine (thrashing / exceeds cache / can't run)

iii. Constraints:
   - Application: particles/processor (N-body), transactions/processor (distributed DB)
   - Resources: time-limited, memory-limited, problem-limited (same problem faster)

Amdahl's Law:

$$S_p(n) = \frac{T_*(n)}{f \cdot T_*(n) + \frac{1-f}{p} T_*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

i. Sequential fraction: $0 \leq f \leq 1$ limits speedup

ii. Discourage from making large parallel computers, focus on reducing sequential fraction f

iii. Rebuttal: circumvented for large problem size
   - Commonly f is a function of n, $f(n)$
   - Efficient parallel algorithm: $\lim_{n\to\infty} f(n) = 0$
   - Thus, $\lim_{n\to\infty} S_p = \frac{p}{1+(p-1)f(n)} = p$

Gustafson's Law:

$$S_p(n) = \frac{\tau_f + \tau_v(n,1)}{\tau_f + \tau_v(n,p)}$$

i. Encouraged to use more processors to solve larger problem in the same time (in CPU-bound apps)

ii. τ_f: constant sequential execution time

iii. $\tau_v(n,p)$: parallel execution time on p processors; if perfectly parallelizable (no overheads):
   - $\tau_v(n,1) = T^*(n) - \tau_f$
   - $\tau_v(n,p) = \frac{T^*(n) - \tau_f}{p}$

iv. If $T^*(n)$ increases strongly monotonically with n, then $\lim_{n\to\infty} S_p(n) = p$

Communication

Communication Performance Measures:

i. Bandwidth, B: maximum data rate (bytes/s)
ii. Throughput: effective bandwidth (bytes/s)
iii. Time of Flight, T_{delay}: time for first bit to arrive
iv. Byte Transfer Time, t_B: time to transmit one byte
v. Transmission Time: time to transmit message
vi. Transport Latency: time to transfer a message (time of flight + transmission time)
vii. Sender Overhead, O_{send}: time to compute checksum, append header, and execute routing
viii. Receiver Overhead, O_{recv}: time to compare checksum and generate acknowledgement

Communication Time (message size m):

$$\begin{aligned}T(m) &= O_{send} + T_{delay} + \frac{m}{B} + O_{recv} \\ &= T_{overhead} + \frac{m}{B} \\ &= T_{overhead} + t_B \cdot m\end{aligned}$$

where:

i. $T_{overhead} = O_{send} + T_{delay} + O_{recv}$ is independent of message size m
ii. Byte Transfer Time, $t_B = \frac{1}{B}$
iii. Assume no checksum error, and no network contention / congestion

Performance Analysis

Practical Workflow:

i. Measure first; define target metric (latency vs throughput vs speedup)
ii. Localise the bottleneck (CPU vs memory vs sync vs I/O vs network)
iii. Change *one* factor at a time; keep workload and environment fixed
iv. Validate: speedup should be explained by reduced dominant component

Instrumentation

Performance Instrumentation finds performance metrics:

i. Latency: time to service a request
ii. Response Time: wall-clock duration of an execution
iii. Throughput: average work completed per unit time
iv. IOPS: I/O operations per second
v. Utilization: fraction of time a resource is busy
vi. Saturation: degree to which a resource has queued work it cannot service

Perspectives:

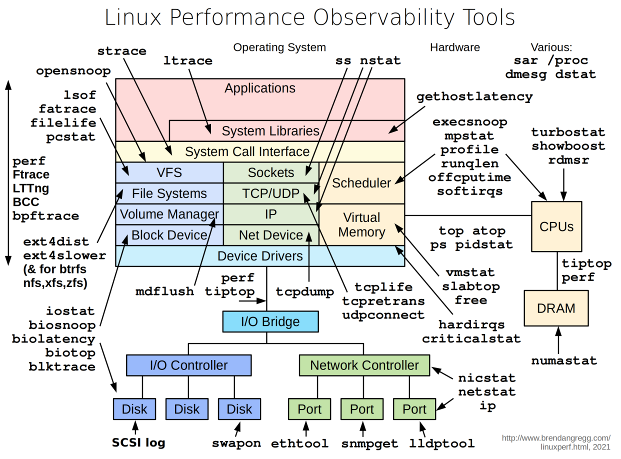
i. Resource Analysis: focus on utilization
   - Time-based: average time the resource was busy
   - Capacity-based: resource throughput
ii. Workload Analysis: focus on throughput, latency

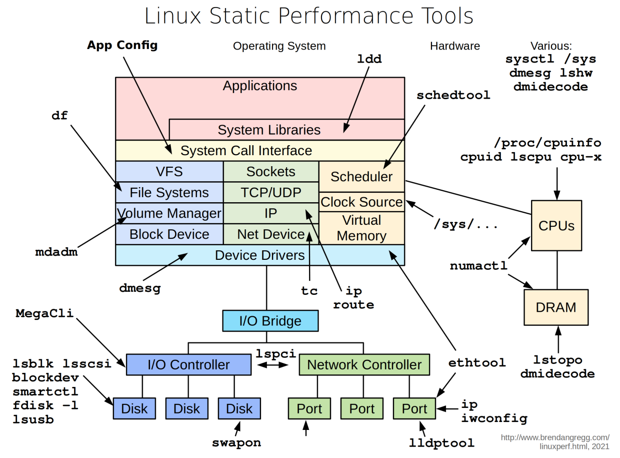
Methodologies:

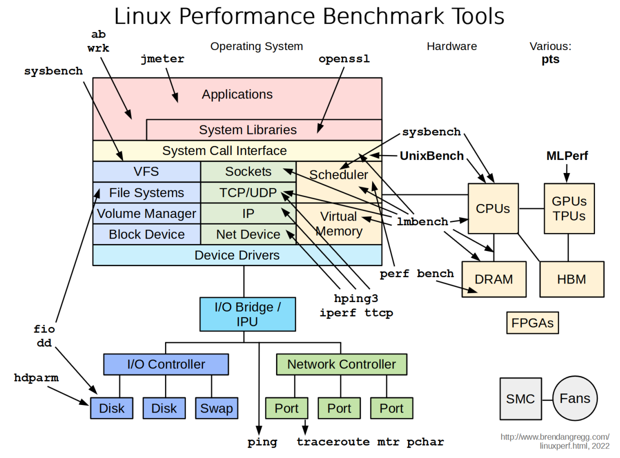
i. Anti-Methodologies: undeliberate, street light (look for obvious issues), drunk man (tune at random)
ii. Problem Statement
iii. USE (per resource): Utilization, Saturation, Errors
iv. Monitoring: record metrics over time

Tool Types & Counters:

i. Observability: watch activity under workload (timing statements, performance counters)
ii. Static: examine system at rest
iii. Benchmarking: load test (caution: contention/production risk)
iv. Tuning: change defaults (danger: regressions now/later under load)
v. Categorization:
   - System-wide vs per-process
   - Fixed counters (kernel-maintained metrics) vs event-based counters (enabled as needed)
   - Profiling: samples/snapshots vs Tracing: record every event occurrence







5. Energy Efficiency

Energy-Efficiency Motivation:

i. Power dissipation has become a limiting factor (*"power wall"*)
ii. Higher performance ⇒ more/faster computers ⇒ more power/heat/cooling/space/cost

Mobile Computing

Trends:

i. Energy-efficiency: decrease power consumption in hardware and increase performance
ii. Increase battery capacity
iii. ARM processing systems:
   - Designed for power-efficiency
   - Closing in on x64 performance
   - Highly customizable architecture

ARM big.LITTLE (low-power approach):

i. Big CPU: high performance for intensive workloads
ii. Little CPU: low power for majority workloads
iii. Switch between cores depending on demand

Heterogeneous Computing

Heterogeneous Platforms:

i. Processors: brawny vs wimpy, big–little
ii. Supercomputers: accelerators
iii. Data Centers: mixed server generations
iv. Cloud: heterogeneous resources with different price–performance

Challenges:

i. Use heterogeneity to reduce power while maintaining performance
ii. Energy-efficient configuration for parallel applications is complex (scheduling)
iii. Burden shifts to programmers for portable code

Data Centers:

i. Large-scale data centers consume significant electricity and emit substantial CO_2
ii. Cooling overhead can be comparable to computation energy
iii. Power Use Efficiency (PUE) $= \frac{\text{Total Energy}}{\text{IT Energy}}$: measures energy efficiency in data centers

Google (energy-efficiency practices):

i. Continuously measure efficiency:
   - Computation (IT) energy
   - Overhead: cooling and conversions
ii. Build custom, highly-efficient servers:
   - Minimize AC/DC conversion losses
   - Remove unnecessary parts
   - Strategic rack placement
   - Tune fan speeds for cooling
iii. Extend equipment lifecycle
   - Reuse, resell components
iv. Control equipment temperature:
   - Raise operating temperature (e.g. 26°C)
   - Thermal modeling
v. Cooling with water instead of chillers:
   - Evaporative cooling, sea water, recycled water

Reducing Energy Consumption:

i. Move less data:
   - Reduce transfers to/from memory
   - Exploit locality
   - Use compression
ii. Use specialized processing:
   - Compute less (no parallel if it increases work)
   - CPU-like cores + GPU-like throughput-optimized cores
   - FPGAs (programmable hardware)

6. Appendix

name	**prefix**	**multiplier**
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	K	10^{3}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}

name	**prefix**	**multiplier**
exbi	Ei	2^{60}
pebi	Pi	2^{50}
tebi	Ti	2^{40}
gibi	Gi	2^{30}
mebi	Mi	2^{20}
kibi	Ki	2^{10}