CS3211 Parallel and Concurrent Prog.

AY 25/26 Sem 2 — github/omgeta

1. Introduction

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

Concurrency vs. Parallelism:

i. Concurrency: tasks overlap in time (may interleave on a single core)
ii. Parallelism: tasks run at the exact same time (simultaneously on different cores)
   - Multiple cores, with each core supporting multiple hardware threads (SMT)
   - Depends on available hardware threads

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)

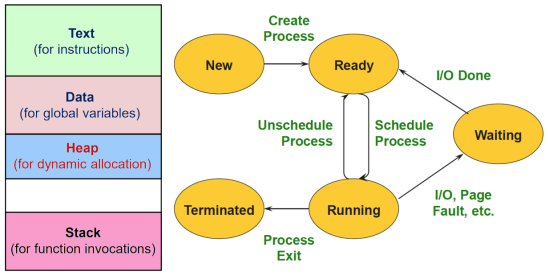
Work Distribution Patterns:

i. Task Parallelism: divide work into different tasks; threads become specialists for specific tasks
ii. Data Parallelism: divide data among threads, runing the same operation on different partitions

Processes

Process is an abstraction for a running program:

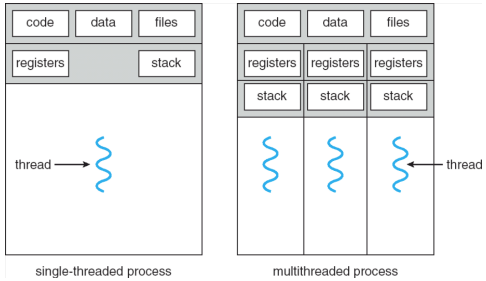
– High system call overhead
– OS allocates and initializes data structures
– Communication goes through the OS



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



Implementations:

i. User-Threads are managed by a userspace library:
   \+ Fast: context switches have low overhead
   – Cannot parallelize to different cores
   – 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

Synchronisation

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

Race Condition is when two concurrent threads access a shared resource without synchronisation, and at least one thread modifies the shared resource.

Data Race is when two concurrent memory accesses target the same location, with both not being reads or sync operations.

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

Mechanisms:

i. Lock: primitive acquire(), release()
ii. Semaphore: atomic counter ≥ 0; wait(), signal()
iii. Monitors: thread-safe high-level data structures
iv. Messages: explicit sync with atomic data transfer

2. C++

Threads

Operations:

i. Construct:
   - Function: `std::thread t(f, args...)`
   - Callable object: `std::thread t(F{}, args...)`
   - Lambda: `std::thread t([]{ ... })`

ii. Wait Once: `t.join()`
   - Check with `t.joinable()`

iii. Detach: `t.detach()`
   – Extra care with local variables: detached thread may outlive their scope (dangling references).

iv. Pass Args By Reference: `std::ref(x)`

v. Move Ownership (cannot copy): `m2 = move(m)`

Mutexes

Operations:

i. Construct: `std::mutex m`

ii. Explicit: `m.lock()`, `m.unlock()`
   - Multiple (avoid deadlocks): `std::lock(m1, m2, ...)`
   – Must unlock on every exit path, incl. exceptions

iii. RAII: `std::lock_guard<std::mutex> g(m)`
   - Adopt Existing Locked Mutex: `std::lock_guard g(m, std::adopt_lock)`
   - Multiple (avoid deadlocks): `std::scoped_lock lk(m1, m2, ...)`
   + Locks on construction and unlocks on scope exit

iv. Flexible: `std::unique_lock<std::mutex> lk(m)`
   - Adopt Existing Locked Mutex: `std::unique_lock lk(m, std::adopt_lock)`
   - Defer Lock Until Later: `std::unique_lock lk(m, std::defer_lock)`
   + RAII unlocks on scope exit if mutex owning
   + Allows multiple lock/unlock within a scope
   + Needed for condition variables

Condition Variables

Condition variables sync actions without busy-waiting.

Operations:

i. Construct: `std::condition_variable cv`
   - Works only on `std::unique_lock<std::mutex>`
   - Generic (any mutex-like, less efficient): `std::condition_variable_any cva`

ii. Wait: `cv.wait(lk)` / `cv.wait(lk, pred)`
   - Unlocks while blocking, then locks before return
   – Spurious Wakeups: predicate may be checked any number of times; needs no side effects

iii. Wake One Thread: `cv.notify_one()`

iv. Wake All Threads: `cv.notify_all()`

Monitors

Monitors encapsulate shared data and operations enforced by a mutex and condition variables.

i. One thread executes monitor operations at a time

ii. Threads wait for conditions and state changes

Example:

```
class BoundedBuffer {
  std::mutex m;
  std::condition_variable cv;
  std::queue<int> q;
  size_t K;

public:
  void push(int x) {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [&]{ return q.size() < K; });
    q.push(x);
    cv.notify_one();
  }

  int pop() {
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [&]{ return !q.empty(); });
    int x = q.front(); q.pop();
    cv.notify_one();
    return x;
  }
};
```

Memory Model

Memory model is the contract to programmers about how memory operations will be reordered by the compiler.

i. Indep. read/write may be reordered to optimize

ii. Ordering Relations:
   - Sequenced-Before: order within a single thread
   - Synchronizes-With: cross-thread with sync ops
   - Happens-Before: transitive visibility relation of Sequenced-Before ∪ Synchronizes-With
   - Modification Order: order of writes per atomic

iii. SC-for-DRF: modern compilers guarantee data race free programs behave as if sequentially consistent
   - Guaranteed if every atomic uses `seq_cst`

Atomics (`std::atomic<T>`):

i. Load: `a.load(std::memory_order)`

ii. Store: `a.store(val, std::memory_order)`

iii. May use internal locks (not always faster)
   - `a.is_lock_free()`/ `T::is_always_lock_free`

Memory Models:

i. Sequentially Consistent (`seq_cst`)

ii. Relaxed (`relaxed`):
   - No *synchronizes-with*; minimal cross-thread ordering guarantees
   - Still atomic; participates in modification order for that atomic variable

iii. Acquire-Release (`acquire`, `release`, `acq_rel`)
   - If an acquire load reads-from a release store on the same atomic ⇒ *synchronizes-with*
   - Writes before the release become visible after the acquire (in the acquiring thread)

Fences (`std::atomic_thread_fence(memory_order)`):

i. Enforces ordering without modifying data

ii. Typically constrain reorder around relaxed atomics

Concurrent Data Structures

Concurrent data structures let multiple threads access it safely without race conditions or broken invariants.

Granularity:

i. Coarse: one mutex protects the whole structure
   - Simple + safe, but serializes all operations

ii. Fine: multiple mutexes protect indep. parts
   - More true concurrency, but higher complexity

iii. Reader/Writer Split: allow concurrent reads, exclusive writes (e.g. `std::shared_mutex`).

Lock-based Examples:

i. Stack (one global mutex):
   + Safe for concurrent calls, exception-safe
   – Work is serialized
   – No built-in waiting for items: caller must poll `empty()` or catch `empty_stack` exceptions

ii. Queue (one mutex + condvar):
   + `wait_and_pop()` blocks until an item exists
   – Issue: `notify_one()` on thread can throw
      - Fixes: `notify_all()` (costly), `notify_one()` on exception, use shared pointers

iii. Queue (`std::shared_ptr<T>` node):
   + Ensures data deleted when no longer needed
   + Better exception safety; shorter lock hold time
   – Still serialized by one global mutex

iv. Queue (fine; dummy node + separate locks):
   - Add dummy node so if empty `front == back`
   - Avoid race on the same node: ensure `front->next` and `back->next` are not the same
   + Separate `front_mutex` / `back_mutex` ⇒ concurrent `push()` and `pop()`
   – More complex invariants and locking discipline

v. Queue (super-fine; per-node locks):
   + Synchronizes-with push and pop threads
   – Correctness complexity increases (avoid UAF; lock ordering becomes subtle)

Nonblocking Data Structures use no blocking library calls:

i. Obstruction-free: any given thread completes in bounded steps if others are paused

ii. Lock-free: if multiple threads are operating, some thread completes in bounded steps
   - Threads need not do same operations
   - If a thread is suspended, other threads must still be able to complete without waiting
   + Maximum concurrency: some thread makes progress every step
   + Robust: if a thread dies mid-operation, only that thread's data is lost; others proceed
   – Livelock possible: two threads repeatedly restart because each sees the other's changes
   – Can reduce overall performance even if individual waiting time drops
   – Atomics can be much slower than non-atomic ops (hardware sync on shared atomics)
   – Memory contention / write propagation ⇒ cache ping-pong under contention
   – False sharing can also cause cache ping-pong (harder to identify)

iii. Wait-free: if multiple threads are operating, every thread completes in bounded steps
   - Must be starvation-free
   - Algorithms with unbounded retries because of clashes with other threads are not wait-free

Guidelines:

i. Prototype with `std::memory_order_seq_cst`

ii. Use a lock-free memory reclamation scheme:
   - Track how many threads reference an object; delete when no longer referenced
   - Recycle nodes

iii. Watch out for ABA:
   - Include an ABA counter alongside the variable

iv. Identify busy-wait loops and help the other thread

3. Go

Goroutines

Goroutines are lightweight function tasks running in the same address space.

Operations:

i. Start: `go f(args...)`
ii. Lifetime:
- Program exits when `main` returns
- Not GC'd: avoid goroutine leaks

iii. Blocks only block the thread, no other goroutines
iv. Preemptable: by Go's runtime

Channels

Channels coordinate goroutines by communication.

Operations:

i. Make: `ch := make(chan T)` / `make(chan T, C)`
- Directional: `chan<- T` (send-only), `<-chan T` (recv-only)

ii. Send: `ch <- v`
- Blocks if channel is full (buffered) / until a receiver (unbuffered)

iii. Receive: `v, ok := <-ch`
- Blocks if channel is empty (can cause deadlocks)
- `ok` false $\iff$ `v` is default from closed channel

iv. Close: `close(ch)`
- Allowed from closed channel any number of times
- Sending to a closed channel panics

v. Range: `for v := range ch { ... }` runs until channel is closed and drained

Ownership Pattern:

i. Owner goroutine makes, writes, and closes channel
ii. Expose only `<-chan T` to consumers

select

`select` composes channel operations:

i. Blocks if no case is ready (unless `default`)
ii. If multiple cases are ready: runtime picks pseudo-randomly (uniform over ready cases)
iii. Stopping:
- `time.After(d)`: channel that fires after `d`
- `done` channel to signal stop

For-select Loop:

```
for {
  select {
  case <-done:
    return
  default:
    // do work
  }
}
```

sync Package

Use mostly in small scopes (e.g., inside a `struct`); prefer higher-level coordination with channels when possible.

i. WaitGroup: `var wg sync.WaitGroup`
- `wg.Add(n)`, `wg.Done()`, `wg.Wait()`, `wg.Go(f)`

ii. Mutex: `var mu sync.Mutex`:
- Locks: `mu.Lock()`, `mu.Unlock()`

iii. Read-Write Mutex: `var rw sync.Mutex`:
- Reads: `rw.RLock()`, `rw.RUnlock()`
- Writes: `rw.Lock()`, `rw.Unlock()`

iv. Cond: `c := sync.NewCond(&mu)`
- Wait: `c.Wait()` unlocks & blocks, then re-locks before returning
- Wake: `c.Signal()` / `c.Broadcast()`

v. Once: `once.Do(f)` runs `f` at most once
vi. Pool: `p.Get()` / `p.Put(x)`

Memory Model

Go's memory model defines when a read in one goroutine is guaranteed to observe a write from another.

i. Happens-Before: transitive visibility relation of Sequenced-Before $\cup$ Synchronizes-With
ii. Within goroutine: observe program order (Sequenced-Before)
iii. Across goroutines: observed order may differ without synchronization
iv. Guarantee a read r of variable v observes write w if:
- w happens before r
- Other write to v happens before w or after r

Concurrency Patterns

Patterns:

i. Confinement:
- Ad-hoc: only one goroutine mutates shared data
- Lexical: restrict access to shared locations (e.g., expose only `<-chan T`, slice of array)

ii. Prevent goroutine leaks:
- Creator ensures termination when work done, unrecoverable error, or when told to stop working
- Beware nil channels: read/write blocks forever

iii. Error handling:
- Return result + error along the same channel
- Centralize decisions in a state-goroutine

iv. Pipeline:
- Stages connected by channels; each stage is a group of goroutines running the same function

v. Fan-out / Fan-in:
- Fan-out: replicate a slow stage across goroutines (independent work; order not guaranteed)
- Fan-in: multiplex multiple input channels into one output channel

vi. Load balancing:
- Pool of workers reads from a shared work channel
- Each worker emits results; collect with fan-in

4. Classical Synchronisation Problems

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);
```

Sleeping Barber

Barber sleeps until a customer wakes him. If barber is busy and chairs are free, the customer sits or will leave if there are no chairs.

```
// Customer
wait(mutex);
if (customers==n){
  signal(mutex);
  exit();
}
customers += 1;
signal(mutex);

signal(customer);
wait(barber);
getHairCut();
signal(custDone);
wait(barbDone);

wait(mutex);
customers -= 1;
signal(mutex);
```

```
// Barber
while (1) {
  wait(customer);
  signal(barber);
  cutHair();
  wait(custDone);
  signal(barbDone);
}
```

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);
  }
}
```

5. Rust

Ownership and Borrowing

Rust enforces memory-safety at compile-time without GC, preventing data races and enabling concurrency.

i. Each value has exactly one owner
+ Ownership prevents double-free; owner frees
+ Borrowing prevents use-after-free (UAF)
+ Segfault-free

Operations:

i. Move: `T`
   - Transfers ownership; old binding cannot be used
   - Deep Copy: `clone()` is explicit (expensive)
ii. Shared Borrow: `&T`
   - Both bindings are read-only while shared
iii. Mutable Borrow: `&mut T`
   - Old binding is read-only while shared
   - New binding is mutable while shared

Threads

Operations (`std::thread`):

i. Spawn: `let h = spawn(move || {...})`
   - `move`: moves captured values into thread
ii. Join: `h.join()`
iii. Marker Traits:
   - `Copy`: safe to memcpy (i.e. `u32`, `f32`, no String)
   - `Send`: safe to transfer to another thread
   - `Sync`: safe to share reference between threads; `T` is `Sync` $\iff$ `&T` is `Send`
iv. Shared Ownership:
   - `Rc<T>`: within a thread
   - `Arc<T>`: across threads; has `Send` trait

Mutexes

Operations (`std::sync::Mutex`):

i. Construct: `let m = Mutex::new(x)`
ii. Lock-Guard: `let mut g Guard<T> = m.lock()`
   - Guard controls access: `*g += 1`
   - When guard goes out of scope, mutex unlocks

Atomics

Atomics are guaranteed lock-free indivisible types.

Operations (`std::sync::atomic`):

i. Construct: `let a = AtomicUsize::new(v)`
   - `AtomicBool`, `AtomicIsize`, `AtomicI16`
ii. Load: `a.load(SeqCst)`
iii. Store: `a.store(v, SeqCst)`
iv. Read-Modify-Write: `a.fetch_add(1, SeqCst)`, `a.swap(v, SeqCst)`

Multi-Producer, Single Consumer (MPSC)

MPSC provides a FIFO communication queue.

Operations (`std::sync::mpsc`):

i. Create: `let (tx, rx) = channel()`
ii. Clone Sender: `let tx2 = tx.clone()`
iii. Send: `tx.send(v)`
iv. Receive: `rx.recv()`

crossbeam Package

Crossbeam extends beyond `std`.

Operations:

i. Scoped Threads: `crossbeam::scope(...)`
   - Threads within borrow only from the scope
   - All spawned threads must end before scope exits
ii. MPMC Channels: `crossbeam::channel()`
iii. Bounded Channels: `crossbeam::bounded(C)`
iv. Backoff: `crossbeam::utils::Backoff::new()`
   - Exponentially slow retry reduces contention, spin

```
let mut v = vec![1, 2, 3, 4];
thread::scope(|s| {
  let (a, b) = v.split_at_mut(2); // disjoint
  s.spawn(|_| { for x in a { *x += 1; } });
  s.spawn(|_| { for x in b { *x += 1; } });
}).unwrap();
println!("{:?}", v); // safe: both returned
```

rayon Package

Rayon provides data-parallelism operations.

Operations (`rayon::prelude::*`):

i. Iterators:
   - Read-Only: `xs.par_iter()`
   - Mutable: `xs.par_iter_mut()`
ii. Combinators:
   - `map(...)`, `filter(...)`, `for_each(...)`
   - Aggregation: `reduce(...)`/ `sum()`
iii. Sort: `xs.par_sort()`/ `xs.par_sort_unstable()`

```
let max = AtomicI64::new(MIN);
vec.par_iter().for_each(|n| {
  loop {
    let old = max.load(SeqCst);
    if *n <= old { break; }
    let r = max.compare_and_swap(old, *n, SeqCst);
    if r == old {
      // swapped
      break;
    }
  }
})
```

Non-blocking I/O

Non-blocking I/O enables concurrency with one thread:

i. Blocking I/O: thread sleeps until data arrives
ii. `epoll`: kernel reports ready file descriptors
iii. Event Loop: wait for ready fds → handle I/O

Futures

Futures represent work that will produce a value later:

i. `Future`: trait driven by `poll(Context)`:
   - Runs until it can no longer make progress
   - Returns `Poll::Ready(T)` or `Poll::Pending`
ii. `Context` provides `wake()` (notify executor to repoll)
iii. Combinators: `.then()`

Executors are user-space schedulers that drive futures:

i. `Pending` ⇒ future arranges a `wake()` when it can progress again
ii. If nothing can progress: sleep until a `wake()`
iii. Threading with Tokio:
   - Single-thread: one OS thread polls all futures
   - Multi-thread: futures may run on multiple cores
   - Shared data still needs synchronization

Async/Await

Async/Await are syntactic sugar for composing Futures.

i. `async fn` returns a `Future`
ii. `.await` waits for a future (only inside `async`)
\+ Zero-cost Abstraction: compiler lowers `async fn` into a state machine implementing `poll()`
\+ Lower memory use with stackless coroutines
\+ Lower scheduling overhead from context switches
\+ Scales to very high concurrency (esp. I/O bound)
– Block inside async sleeps the executor thread
– Cooperative: tasks yield only at `.await`
– CPU-heavy work can starve other tasks
– No recursion; future size fixed at compile time

6. Testing

Bug Types:

i. Unwanted Blocking: deadlock, livelock, IO
ii. Race Conditions:
   - Data races ⇒ undefined behavior (unsynchronized concurrent access)
   - Broken invariants, partial updates observed
   - Dangling pointers / use-after-free across threads
   - Memory corruption, double-free
   - Lifetime issues (thread outlives data it accesses)

Techniques:

i. Review the code:
   - Identify shared data and how it is protected
   - Track held mutexes and find lock ordering issues
   - Check cross-thread ordering enforcement
   - Validate lifetimes (e.g. `join()` should not skip on exceptions)
ii. Testing:
   - Run the smallest amount of code that can demonstrate the problem
   - Remove concurrency to confirm it is concurrency-related
   - Run on multicore and (when useful) single-core

Tools:

i. Valgrind memcheck (memory error; high overhead): `valgrind --tool=memcheck <prog ...>`
ii. Helgrind/DRD (thread error detectors):
   - Detects misuses of pthreads, potential deadlocks (lock ordering), data races
   - Uses lock-order graphs, happens-before reasoning
iii. Sanitizers (compile-instrumentation; mid overhead):
   - ASan: `-fsanitize=address` (addressability)
   - TSan: `-fsanitize=thread` (data races, thread/lifetime)

Model Checking

Check Formal Specifications:

i. Build a model in a special DSL / new language
ii. Check: constraints, unexpected behaviour, deadlock
iii. Why spend time:
   - Check designs before costly implementation
   - Prove properties for existing code
   - Enable aggressive optimizations without breaking correctness
\+ Rigorous, check all traces exhaustively
\+ Produces trace violating requirement
– Tedious to define correct specification

Approaches:

i. Write a formal specification and check it (with model checker/ proof assistant)
ii. Use specification to write minimal code:
iii. Add invariants as comments to assist checker:

Model Checkers:

i. TLA+ (TLC): temporal properties; good for concurrent/distributed systems
   - Write a TLA+ specification; use TLC to exhaustively explore the state space
   - Check safety (invariants) and liveness (temporal)
   - Reports errors with a counterexample trace
   - Model Structure: `Init` (initial states) + `Next` (state transitions)
   - Uses fairness to rule out infinite "stuttering" (non-progress) executions
ii. Coq: interactive proofs; generate code (OCaml/Haskell/Scheme)
iii. Alloy: relational logic; good for modeling structures

Challenges of Distributed Systems:

i. Reliability: in case of failures
ii. No global clock or ordering: need memory model
iii. Consistency
iv. Consensus