PProg Speedrun (FS25) - Shivram Sambhus @ cs.shivi.io

1 TOC

2 Introduction & Performance Models

The End of the Free Lunch

For decades, software performance improved "for free" as single-core CPU clock speeds increased. This trend has stopped due to physical limitations (power consumption, heat dissipation). The industry's response was to add more cores to chips. This means performance gains now come from **parallelism**, a burden shifted from hardware engineers to software developers.

Concurrency vs. Parallelism

- **Concurrency:** A program is concurrent if it can manage multiple logical tasks over overlapping time periods. It's a **structural** concept. Think of a web server handling multiple client requests. On a single core, it achieves this by interleaving execution (context switching).
- **Parallelism:** A program is parallel if it executes multiple tasks simultaneously. It's an **execution** concept that requires multiple hardware cores. Think of rendering a video where different cores process different frames at the same time. Parallelism is a way to implement concurrency.

Amdahl's Law: The Pessimist's View (Fixed Workload)

Big Idea: The sequential part of your program is an inescapable bottleneck that limits your maximum speedup.

Formula: Given a fraction f of a program that is sequential, the speedup S on P processors is limited by:

$$S_p \leq \frac{1}{f + \frac{1-f}{P}}$$

Intuition: Imagine a 10-hour task. 2 hours (20%, $f = 0.2$) are sequential (e.g., reading input, final report generation). The other 8 hours are parallelizable. Even with infinite processors, the parallel part takes 0 time, but the 2 sequential hours remain. The total time can never be less than 2 hours. The maximum speedup is $\frac{1}{0.2} = 5x$. This law governs tasks with a **fixed workload**.

Gustafson's Law: The Optimist's View (Fixed Time)

Big Idea: When we get more processors, we don't just solve the same problem faster; we solve a **bigger** problem in the same amount of time.

Formula: The scaled speedup is:

$$S_p \leq f + P(1 - f)$$

Intuition: Imagine you have 1 hour to run a weather simulation. On 1 core, you can simulate a 100km grid. On 100 cores, you don't run the 100km simulation in under a minute; you run a 1000km grid simulation with higher resolution in that same hour. This law governs tasks with a **variable workload** that scales with the number of processors.

Task Graphs: Visualizing Parallelism

A Directed Acyclic Graph (DAG) is the best way to model a parallel computation.

- **Nodes:** Tasks (units of work).
- **Edges:** Dependencies (task A must finish before task B can start).
- **Work (T_1):** The sum of the weights of all nodes. It's the time to run on a single processor.
- **Span (T_∞):** The weight of the longest path in the DAG (the **critical path**). It's the minimum possible execution time, no matter how many processors you have, because it represents the fundamental sequential dependency chain.
- **Parallelism:** The ratio $\frac{T_1}{T_\infty}$. This is a crucial metric: it tells you the maximum possible speedup you can hope to achieve. A high parallelism value means the program has a lot of potential to scale.

3 Java Threads & Synchronization

Threads and Race Conditions

A thread is an independent path of execution within a process. All threads in a process share the same memory (heap), which is both powerful (easy data sharing) and dangerous (risk of data corruption).

- **Race Condition:** Occurs when the correctness of a program depends on the unpredictable timing or interleaving of threads.
- **Critical Section:** The block of code that accesses the shared resource and must be executed atomically to prevent race conditions.

Example: `count++` is not one operation. It's three: `read` the value of `count` into a register, `increment` the register, and `write` the register's value back to `count`. If two threads execute this concurrently, they can both read the same initial value, and one thread's write will be lost.

`synchronized` and Intrinsic Locks

Java's solution is the **monitor**, or **intrinsic lock**. Every Java object has one. The `synchronized` keyword acquires this lock.

- **Mutual Exclusion:** Only one thread can hold a given monitor at a time. Any other thread attempting to acquire it will be **BLOCKED**.
- **Reentrancy:** A thread that already holds a lock can acquire it again without deadlocking. The JVM keeps a per-thread count; the lock is only released when the count reaches zero.

```
public class Counter {
    private long count = 0;
    // The lock of `this` is acquired upon entry
    // and released upon exit.
    public synchronized void increment() {
        count++;
    }
}
```

Coordination with `wait()`, `notify()`, `notifyAll()`

These methods allow threads to pause execution and release a lock until a condition is met, preventing inefficient busy-waiting. Used for Producer-Consumer patterns.

The Rules:

1. You **must** own the monitor of the object you are calling `wait()` or `notify()` on.
2. You **must** call `wait()` inside a `while` loop that checks the condition. This is to guard against **spurious wakeups** and to ensure the condition is still true after re-acquiring the lock.

```
// Consumer waiting for an item in a shared queue
public synchronized String take() throws InterruptedException {
    // The while loop is the "condition predicate"
    while (queue.isEmpty()) {
        // Releases the lock on `this` and waits.
        wait();
    }
    return queue.poll();
}

// Producer adding an item
public synchronized void put(String item) {
    queue.add(item);
    // Wakes up ONE waiting consumer.
    notify();
}
```

4 Software Lock Implementations

The Challenge: Locks from `volatile`

Before hardware support like CAS, computer scientists designed locks using only the guarantee of atomic reads and writes to memory, which `volatile` provides in Java. `volatile` ensures that writes are immediately visible to other threads and prevents compiler instruction reordering.

Decker's Algorithm (2 Threads)

A polite but complex negotiation. "I want to enter. If you also want to enter, we'll check whose `turn` it is. The person whose turn it isn't must yield by lowering their flag, wait, and then try again."

```
// Shared variables for Decker's Algorithm
volatile boolean want0 = false, want1 = false;
volatile int turn = 0;

// Code for Thread 0
public void lock0() {
    want0 = true;
    while (want1) {
        if (turn == 1) {
            want0 = false; // Politely yield
            while (turn == 1) { /* spin */ }
            want0 = true; // Re-assert interest
        }
    }
}
public void unlock0() {
    turn = 1; // Give turn to the other thread
    want0 = false;
}

// Code for Thread 1 is symmetric
```

Peterson's Lock (2 Threads) - A Simpler Approach

The Idea: A more elegant negotiation. "I want to enter. To be polite, I'll also declare myself the `victim`, effectively saying 'you go first'."

Components:

- `volatile boolean[] flag`: `flag[i]` is true if thread `i` wants to enter.
- `volatile int victim`: The ID of the thread that must wait.

Intuition: A thread `i` only waits if the **other** thread `j` **also** wants to enter (`flag[j]`) AND thread `i` was the **last one** to set the victim variable (`victim == i`).

```
public class PetersonLock {
    private volatile boolean[] flag = new boolean[2];
    private volatile int victim;

    public void lock(int i) {
        int j = 1 - i;
        flag[i] = true;      // I want in
```

```
        victim = i;          // But you go first
        while (flag[j] && victim == i) { /* spin */ }
    }
    public void unlock(int i) {
        flag[i] = false;     // I'm done
    }
}
```

Properties: Guarantees Mutual Exclusion, is Deadlock-free, and Starvation-free (Fair) for 2 threads.

Filter Lock (N Threads)

The Idea: Generalizes Peterson's Lock to N threads by creating N-1 "waiting levels" that a thread must pass through. Only one thread can be at the final level, granting it access to the critical section.

Mechanism:

- To enter level `L`, thread `me` sets `level[me] = L` and declares itself the `victim[L]`.
- It then waits as long as there is **any other thread** at a level greater than or equal to `L` **and** `me` is still the victim at level `L`.

```
public class FilterLock {
    private volatile int[] level; // level[i] for thread i
    private volatile int[] victim; // victim[L] for level L
    private final int n;

    public FilterLock(int n) {
        this.n = n;
        level = new int[n];
        victim = new int[n];
    }

    public void lock(int me) {
        for (int L = 0; L < n - 1; L++) {
            level[me] = L;
            victim[L] = me;
            // Spin while there's another thread at my
            // level or higher, AND I am the victim.
            boolean conflict;
            do {
                conflict = false;
                for (int k = 0; k < n; k++) {
                    if (k != me && level[k] >= L) {
                        conflict = true;
                        break;
                    }
                }
            } while (conflict && victim[L] == me);
        }
    }

    public void unlock(int me) {
        level[me] = -1; // No level
    }
}
```

Properties: Guarantees Mutual Exclusion and is Deadlock-free for N threads. It is **not** Starvation-free. A thread can be stuck at a level if other threads keep entering that level and making it the victim.

Lamport's Bakery Algorithm (N Threads)

The Idea: A bakery queue. Each thread takes a numbered ticket. The thread with the lowest ticket gets served (enters the critical section) next. Tie-breaking is done by thread ID.

Mechanism:

1. **Take ticket:** Thread `i` sets `choosing[i] = true`, picks `ticket[i] = max(all tickets) + 1`, then sets `choosing[i] = false`.
2. **Wait:** For every other thread `j`, wait until `choosing[j]` is false. Then, wait as long as `j` has a "better" ticket: `(ticket[j], j) < (ticket[i], i)`.

```
public class BakeryLock {
    private volatile boolean[] choosing;
    private volatile int[] ticket;
    private final int n;

    public BakeryLock(int n) {
        this.n = n;
        choosing = new boolean[n];
        ticket = new int[n]; // Defaults to 0
    }

    public void lock(int i) {
        // 1. Announce that I am choosing a ticket
        choosing[i] = true;
        // Find the highest ticket number and pick one higher
        int maxTicket = 0;
        for (int t : ticket) {
            maxTicket = Math.max(maxTicket, t);
        }
        ticket[i] = maxTicket + 1;
        choosing[i] = false;

        // 2. Wait my turn in the queue
        for (int j = 0; j < n; j++) {
            if (j == i) continue;
            // Wait for thread j to finish choosing
            while (choosing[j]) { /* spin */ }
            // Wait if j is contending (ticket > 0) AND has
            // a better ticket (lower number or lower ID)
            while (ticket[j] != 0 &&
                   (ticket[j] < ticket[i] ||
                   (ticket[j] == ticket[i] && j < i))) {
                /* spin */
            }
        }
    }

    public void unlock(int i) {
        ticket[i] = 0; // "Return" the ticket
    }
}
```

Why `choosing` flag? It prevents a race condition where two threads might read the current max ticket number simultaneously and pick the same new number. It acts as a lock on the ticket selection process itself.

Properties: Guarantees Mutual Exclusion, Deadlock-free, and Starvation-free (Fair) for N threads.

5 Advanced Synchronization & Deadlocks

Semaphores: The Bouncer

The Idea: A semaphore is a generalized lock that manages a number of permits. It's like a bouncer at a club with a fixed capacity.

- `new Semaphore(N)`: Creates a bouncer for a club with capacity N.
- `acquire()`: Decrements permits. If permits become negative, the thread blocks.
- `release()`: Increments permits and unblocks a waiting thread if any.

A `Semaphore(1)` is a binary semaphore, which acts like a non-reentrant lock.

Example: Bounded Buffer

A bounded buffer is a classic producer-consumer problem where a fixed-size buffer is shared between producers (who put items in the buffer) and consumers (who take items out). Semaphores are used to manage the number of empty slots (spaces) and full slots (items) in the buffer.

```
public class BoundedBuffer<T> {
    private final Queue<T> queue = new ArrayDeque<>();
    private final ReentrantLock lock = new ReentrantLock();
    private final Semaphore spaces; // Counts empty slots
    private final Semaphore items;  // Counts full slots

    public BoundedBuffer(int capacity) {
        spaces = new Semaphore(capacity);
        items = new Semaphore(0);
    }

    public void put(T item) throws InterruptedException {
        spaces.acquire(); // Wait for an empty slot
        lock.lock();
        try {
            queue.add(item);
        } finally {
            lock.unlock();
        }
        items.release();  // Signal there's a new item
    }

    public T take() throws InterruptedException {
        items.acquire();  // Wait for an item
        lock.lock();
        T item;
        try {
            item = queue.poll();
        } finally {
            lock.unlock();
        }
        spaces.release(); // Signal there's a new empty slot
        return item;
    }
}
```

Barriers: The Rendezvous

The Idea: A synchronization point where a group of threads must all wait for each other. It's like hikers agreeing to meet at a summit before continuing.

Mechanism (`CyclicBarrier`): Threads call `await()` which blocks until the required number of `parties` have arrived. When the last thread arrives, all waiting threads are released. The barrier can be reused ("cyclic"), which is crucial for iterative parallel algorithms.

```
// A reusable barrier implementation ("Two-Phase Barrier")
public class ReusableBarrier {
    private final int parties;
    private int count = 0;
    private boolean phase = false; // false=gathering, true=releasing

    public ReusableBarrier(int parties) { this.parties = parties; }

    public synchronized void await() throws InterruptedException {
        boolean currentPhase = phase;
        count++;
        if (count == parties) {
            count = 0;
            phase = !currentPhase; // Flip phase
            notifyAll(); // Release all threads
```

```
        } else {
            while (currentPhase == phase) {
                wait(); // Wait for this phase to end
            }
        }
    }
}
```

Reader-Writer Locks

The Problem: A standard lock is too pessimistic if you have many readers and few writers. Readers don't modify data, so they shouldn't have to block each other.

The Solution: A lock with two modes:

- **Read Lock:** Multiple threads can hold it simultaneously.
- **Write Lock:** Exclusive. Only one thread can hold it, and no read locks can be held at the same time.

This improves performance in read-heavy scenarios.

```
public class Stats {
    private final Map<String, Integer> data = new HashMap<>();
    private final ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();
    private final Lock r = rwLock.readLock();
    private final Lock w = rwLock.writeLock();

    public Integer get(String key) {
        r.lock(); // Multiple threads can enter here
        try { return data.get(key); }
        finally { r.unlock(); }
    }

    public void increment(String key) {
        w.lock(); // Only one thread can enter here
        try { data.merge(key, 1, Integer::sum); }
        finally { w.unlock(); }
    }
}
```

Lock Granularity

The scope of data protected by a single lock.

- **Coarse-Grained:** One big lock for the whole data structure (e.g., `synchronized(list)`). Simple to implement, but kills parallelism as only one thread can operate at a time, even on different parts of the structure.
- **Fine-Grained:** Many small locks, each protecting a small piece of data (e.g., one lock per node in a list). Allows much higher parallelism but is far more complex and risks deadlock if lock ordering isn't managed carefully (e.g., hand-over-hand locking).

Deadlocks

A state where threads are blocked, each holding a resource and waiting for a resource held by another.

Classic Example: Lock Ordering. Thread 1: `lock(A); lock(B);` Thread 2: `lock(B); lock(A);`

Solution: Enforce a global, consistent order for acquiring locks (e.g., by object hash code). This breaks the "circular wait" condition required for deadlock.

6 Lock-Free Programming & Correctness

Lock-Free with Compare-and-Swap (CAS)

The Idea: Avoid locks entirely. Use optimistic loops with an atomic CAS instruction.

The Pattern:

1. Read the current value of a shared variable (`current`).
2. Compute the new value (`next`).
3. Use `CAS(variable, current, next)` to atomically update the variable **only if** it still equals `current`.
4. If CAS fails, another thread changed the variable. Loop back to step 1.

Example: Lock-Free Stack `push`

```
public class LockFreeStack<T> {
    private AtomicReference<Node<T>> head = new
AtomicReference<>(null);

    public void push(T value) {
        Node<T> newNode = new Node<>(value);
        Node<T> oldHead;
        do {
            oldHead = head.get(); // 1. Read
            newNode.next = oldHead; // 2. Compute
            // 3. Attempt atomic update
        } while (!head.compareAndSet(oldHead, newNode));
    }
    // pop() is more complex due to ABA problem...
}
```

The ABA Problem & `AtomicStampedReference`

The Problem: A thread reads value A for `oldHead`. It gets preempted. Other threads `pop(A)`, `push(B)`, then `push(A)` back on. The stack head is again A, but it's a **different node** instance. The first thread wakes up, its `CAS(A, ...)` succeeds, but it has corrupted the list because the `next` pointer of the new "A" is wrong.

The Solution: `AtomicStampedReference` pairs a reference with an integer "stamp" (or version). The CAS now checks both the reference **and** the stamp. Each modification increments the stamp, so A -> B -> A becomes (A, v1) -> (B, v2) -> (A, v3). The original thread's `CAS((A, v1), ...)` will now correctly fail.

```
public class LockFreeStackWithStamp<T> {
  private AtomicStampedReference<Node<T>> head =
      new AtomicStampedReference<>(null, 0);

  public void push(T value) {
    Node<T> newNode = new Node<>(value);
    int[] stampHolder = new int[1];
    Node<T> oldHead;
    do {
        oldHead = head.get(stampHolder);
        newNode.next = oldHead;
    } while (!head.compareAndSet(
        oldHead, newNode, stampHolder[0], stampHolder[0] + 1));
  }
}
```

Linearizability: The Gold Standard

The Idea: A simple, powerful correctness condition. It makes concurrent operations behave as if they were atomic.

Definition: A concurrent history is linearizable if each operation appears to take effect **instantaneously** at a single, indivisible point in time—the **linearization point**—which must lie between its invocation and response.

Why it's great:

- **Strictness:** It respects real-time ordering. If op A finishes before op B begins, A **must** be ordered before B.
- **Composability:** A system built from linearizable components is itself linearizable.

Finding Linearization Points:

- **Lock-based queue `add()`:** The write that makes the new element visible.
- **Lock-free stack `push()`:** The successful `CAS` that updates the `head` pointer.

7 Advanced Lock-Free Data Structures

Optimistic & Lazy Synchronization

These are advanced patterns for fine-grained locking, often used in list/tree-based structures.

- **Optimistic:** Traverse the data structure **without acquiring any locks**. When you find the location to modify (e.g., `pred` and `curr` nodes), lock them. Then, **validate** that they are still in the correct state (e.g., `pred.next == curr`). If validation passes, perform the modification. If not, unlock and retry the whole operation.
- **Lazy:** Decouples logical deletion from physical deletion. To remove a node, you first mark it as "logically deleted" (e.g., set a `marked` flag). This is a fast operation. Later, other operations (or a dedicated cleanup thread) can perform the physical unlinking of marked nodes. `contains()` methods must ignore marked nodes. This simplifies `remove()` logic.

Fine-Grained List-Based Set

Implements a sorted linked list using fine-grained locking.

- `contains(x)`: Traverses the list lock-free. Since it's read-only, this is fast. For lazy lists, it must check that a found node is not marked for deletion.
- `add(x)`: Uses the optimistic pattern.
  1. Traverse the list without locks to find `pred` and `curr` where `pred.key < x <= curr.key`.
  2. Lock `pred` and `curr`.
  3. **Validate:** Check that `pred` is not marked and `pred.next` is still `curr`.
  4. If valid and `curr.key != x` (no duplicate), create a new node and insert it between `pred` and `curr`.
  5. Unlock and return. If validation fails, unlock and retry.
- `remove(x)`: Similar to add. Find `pred` and `curr`, lock them, validate, and then modify pointers (`pred.next = curr.next`). In a lazy list, you would simply mark `curr` for deletion.

The Michael-Scott Lock-Free Queue

A classic, high-performance non-blocking queue using two `AtomicReference` pointers, `head` and `tail`.

- **Invariant:** `head` always points to a sentinel/dummy node. The actual first item is `head.next`.
- `enqueue(x)`:
  1. Create a new node for `x`.
  2. Loop: Read `t = tail.get()`.
  3. Try to `CAS` `t.next` from `null` to your new node.
  4. If successful, you have reserved your spot. Swing the `tail` pointer to your new node with `CAS(tail, t, newNode)`. If this CAS fails, another thread already swung it, which is fine.
- `dequeue()`:
  1. Loop: Read `h = head.get()`, `t = tail.get()`, and `first = h.next`.
  2. **Sanity check:** If `h == t`, the queue might be be empty or in an intermediate state. If `first` is null, it's empty; return null. If not, another enqueuer is mid-operation, so help them by trying to swing the `tail` pointer (`CAS(tail, t, first)`) and retry.
  3. If queue is not empty, `CAS` the `head` from `h` to `first`.
  4. If successful, you have dequeued the node `first`. Return its value.

8 Consensus & Transactional Memory

The Consensus Problem

The fundamental problem of distributed computing. A set of processes must agree on a single value among a set of proposed values. A valid consensus protocol must satisfy:

- **Agreement:** No two correct processes decide on different values.
- **Validity:** The value decided upon must have been proposed by some process.
- **Termination:** All correct processes eventually decide on a value.

The famous FLP Impossibility Result shows that in a fully asynchronous system, there is no deterministic algorithm for consensus that can tolerate even a single crash failure.

The Consensus Hierarchy

Atomic primitives can be ranked by their power to solve the consensus problem for N threads. This power is quantified by the **consensus number C**.

- $C = 1$: Atomic Registers (`volatile` read/write). Cannot solve consensus for even 2 threads.
- $C = 2$: `Test-and-Set`, `Fetch-and-Add`, `Swap`. Can solve consensus for up to 2 threads.
- $C = \infty$: `Compare-and-Swap` (CAS), `Load-Link/Store-Conditional` (LL/SC). These are **universal primitives**—they can be used to implement a wait-free algorithm for any number of threads.

This hierarchy tells us that to implement an arbitrary N-thread wait-free object, we need a primitive with a consensus number of at least N. This is why CAS is so critical for modern lock-free programming.

Transactional Memory (TM)

The Idea: Give programmers the power of database transactions for memory operations. This avoids the complexity of locks and lock-free algorithms. Programmers wrap code in an `atomic` block, and the system ensures it runs indivisibly.

How it Works: The system executes the block **speculatively**.

1. A thread begins a transaction, creating a private log of its reads (read-set) and writes (write-set).
2. At the end of the block, it tries to **commit**. It validates its read-set to ensure no other thread has modified those memory locations.
3. If validation succeeds, its write-set is applied atomically. If it fails (a conflict), the transaction is **aborted**, its private changes are discarded, and it retries.

Types: Hardware TM (HTM, e.g., Intel TSX) is fast but limited. **Software TM** (STM) is more flexible but has higher overhead.

Scala-STM: TM in Practice

A mature library for STM.

- Shared state must be wrapped in transactional references, e.g., `Ref`.
- Operations are performed inside an `atomic { implicit txn => ... }` block.
- It provides a powerful `retry` command. If a condition isn't met, `retry` aborts the transaction and parks the thread. The STM runtime will only re-awaken the thread when another transaction modifies a `Ref` that this thread read, making it an elegant alternative to `wait/notify`.

```
def transfer(from: Ref[Int], to: Ref[Int], amount: Int) = {
  atomic { implicit txn =>
    if (from() < amount) retry // Abort and wait
    from() = from() - amount
    to() = to() + amount
  }
}
```

Clock-Based STM Implementation

A common strategy for implementing STM.

- **Global Version Clock:** A single atomic long that is incremented on every successful commit.
- **Transactional Objects:** Each object has a version timestamp.
- **Transaction Logic:**
  1. **Start:** A transaction reads the global clock to get its `read-version`.
  2. **Read:** When reading an object, it checks if `object.version > read-version`. If so, another thread committed a change after this transaction started. The current transaction must abort.
  3. **Commit:** a. Acquire locks on all objects in the write-set. b. Re-validate the read-set one last time. c. If valid, get a new `write-version` by atomically incrementing the global clock. d. Apply changes to write-set objects, updating their versions to `write-version`. e. Release locks.

9 Parallel Frameworks & Distributed Memory

Java Fork/Join Framework

The Goal: Efficiently execute divide-and-conquer algorithms on a pool of threads.

Core Components:

- `ForkJoinPool`: The executor, which manages a pool of worker threads.
- `RecursiveTask<V>` / `RecursiveAction`: The tasks themselves.

The Magic: Work-Stealing. Each worker thread has its own double-ended queue (deque) of tasks.

- A thread gets tasks from the **head** of its own deque (LIFO order, good for locality in recursion).
- When a thread's deque is empty, it becomes a "thief" and steals a task from the **tail** of another, randomly chosen thread's deque (FIFO order, gets big chunks of work).

Why it's brilliant: This provides automatic, decentralized load balancing. Threads blocked on a `join()` can also execute other tasks, preventing deadlocks seen with naive recursion on a fixed thread pool.

Parallel Patterns Revisited

- **Map:** Embarrassingly parallel. Use `ForkJoin` to split the collection and apply the function on each chunk.
- **Reduce:** Use `ForkJoin` to compute reductions on sub-problems and then combine the results up the recursion tree. The combination step is key.
- **Scan (Prefix Sum):** A more complex pattern requiring a two-pass `ForkJoin` algorithm.
  1. **Up-sweep (Reduce):** Traverse up the conceptual tree, computing the sum of each range.
  2. **Down-sweep (Scan):** Traverse down the tree, using the sums computed in the first pass to calculate the correct prefix for each sub-range. The span is $O(\log n)$.

Distributed Memory & MPI

The Model: A collection of processes, each with its own private memory. They cannot directly access each other's memory. Communication happens by explicitly sending and receiving messages over a network. This is the model for supercomputers and clusters.

MPI (Message Passing Interface): The de-facto standard API.

- **Communicator:** A group of processes (e.g., `MPI_COMM_WORLD`).
- **Rank:** The unique ID of a process within a communicator.
- **Point-to-Point:** `MPI_Send(data, dest, ...)` and `MPI_Recv(buffer, source, ...)`.
- **Collective Operations:** Coordinated communication involving all processes.
  - `MPI_Bcast`: One process sends the same data to everyone.
  - `MPI_Reduce`: Everyone sends data to one process, which combines it (e.g., with a sum).
  - `MPI_Allreduce`: A `Reduce` followed by a `Bcast` of the result.
  - `MPI_Scatter`: One process distributes distinct chunks of an array to all processes.
  - `MPI_Gather`: The inverse of `Scatter`.