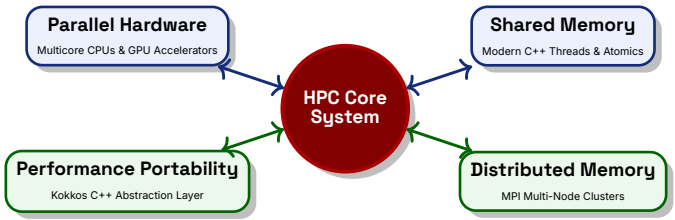
# High-Performance Computing

## Core Concepts in Parallel Systems, Modern C++ & Kokkos



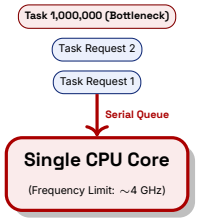

**AHMED EZ-ZOUINE**

**Focus Area:** Parallel Computing · Modern C++ · Kokkos · HPC · Code Optimization

# The Core Challenge: Why Single Cores Cannot Scale

### Why Single Processors Have Hit a Limit

- **Thermal & Power Wall:** Single CPU core clock frequencies plateaued ($\sim 4$ GHz) due to heat dissipation and power density limits.
- **Memory Wall:** Growing latency gap between processor speed and DRAM memory access keeps cores idling.
- **Computational Demand:** Physical simulations, weather modeling, and AI require trillions of operations per second (FLOP/s).




**Sequential Core Bottleneck**

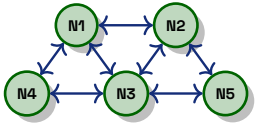
## Practical Impact

A complex fluid dynamics simulation takes **12 years** on a single core, but executing in parallel across cluster nodes reduces runtime to **2 hours**.

# The Solution: Three Ways to Run Code in Parallel

**Three Main Parallel Computing Models**

- **Shared Memory (Threads / OpenMP):**
  - Cores on a single node communicate instantly through a shared memory address space (NUMA).
- **Distributed Memory (MPI):**
  - Compute nodes with isolated memory communicate over high-speed networks (InfiniBand).
- **Accelerators (GPUs / Kokkos):**
  - Thousands of lightweight cores execute data-parallel loops concurrently.




**Interconnected Cluster Network**

## Domain Decomposition

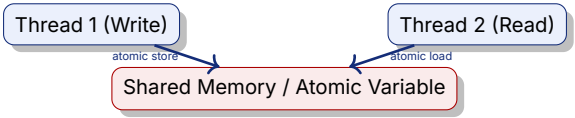
Partitioning large 3D spatial grids into sub-domains so separate processing units compute sub-regions in parallel.

# Multithreading in Modern C++ (Shared Memory)

Executing multiple threads in a shared memory space is fast, but unsynchronized concurrent access creates **Data Races** (corrupted calculations or crashes).

**Thread Synchronization Primitives**

1. **Mutex Locks (`std::mutex`):** Enforces mutual exclusion in critical sections using RAII wrappers (`std::lock_guard`).
2. **Atomic Operations (`std::atomic<T>`):** Lock-free hardware instructions (`fetch_add`) for safe, low-latency state updates.
3. **Modern Threads (`std::jthread`):** C++20 auto-joining threads supporting cooperative cancellation tokens.



# Code Example: Thread-Safe Parallel Accumulation

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>

int main() {
    std::atomic<long long> global_sum{0};
    std::vector<std::thread> workers;

    for (int t = 0; t < 4; ++t) {
        workers.emplace_back([&global_sum, t]() {
            long long local = 0;
            for (int i = 0; i < 1000; ++i)
                local += (t * 1000 + i);
            global_sum.fetch_add(local);
        });
    }

    for (auto& w : workers) w.join();
    std::cout << "Sum: " << global_sum << "\n";
}
```

**Execution Output:**

```
Worker Threads: 4 (1000 items each)
Atomic Total: 7998000
Status: Verification PASSED (Race-Free)
```

## Best Practice

Accumulate values in a thread-local variable first, then perform a single `fetch_add` call at thread exit to minimize hardware bus contention.

# OpenMP: Easy Multithreading for CPU Loops

```cpp
#include <iostream>
#include <vector>
#include <omp.h>

double compute_grid(const std::vector<double>& data,
     int n) {
    double total_sum = 0.0;
    #pragma omp parallel for reduction(+:total_sum)
    for (int i = 0; i < n; ++i) {
        total_sum += data[i] * 2.5 + 1.0;
    }
    return total_sum;
}
```

**Core Advantages**

- **Compiler Directives:** Add `#pragma omp parallel for` above C++ loops without restructuring algorithms.
- **Automated Execution:** Manages thread pools, work partitioning, and reduction accumulation.

**Why OpenMP is Essential**

Provides portable CPU multithreading with minimal code modification overhead.

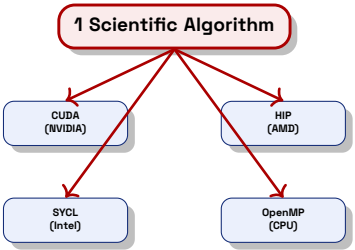
# The GPU Hardware Problem: Vendor Fragmentation

**Hardware Backend Fragmentation**

- **NVIDIA GPUs** require **CUDA**
- **AMD GPUs** require **HIP**
- **Intel GPUs** require **SYCL**
- **Multicore CPUs** require **OpenMP**

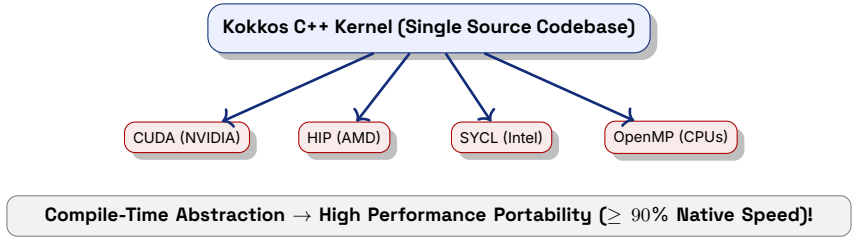
## The Consequence

Rewriting the exact same math in 4 different vendor languages causes code duplication, backend divergence, and high maintenance overhead.



# The Solution: Performance Portability with Kokkos

**What is Kokkos?** An open-source C++ programming model developed by US National Laboratories (SNL, LANL, LBNL).




## Main Benefit

Write algorithms once in standard C++. Kokkos expands code at compile time for NVIDIA, AMD, and Intel GPUs or multicore CPUs while preserving efficiency.

# Kokkos Code Example: Writing a 2D Parallel Loop

```cpp
#include <Kokkos_Core.hpp>

void solve_stencil(int nx, int ny) {
    Kokkos::View<double**> u("u", nx, ny);
    Kokkos::View<double**> f("f", nx, ny);

    using Policy = Kokkos::MDRangePolicy<
                     Kokkos::Rank<2>>;

    Kokkos::parallel_for("stencil_2d",
      Policy({1,1}, {nx-1, ny-1}),
      KOKKOS_LAMBDA(const int i, const int j) {
        f(i,j) = 0.25 * (u(i+1,j) + u(i-1,j) +
                         u(i,j+1) + u(i,j-1));
      });
}
```

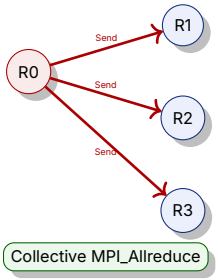
## Core Concepts Explained

- `Kokkos::View`: Multidimensional array selecting optimal memory layout (`LayoutRight`/Row-Major for CPUs vs `LayoutLeft`/Column-Major for GPUs).
- `parallel_for`: Dispatches grid points across parallel execution units.
- `KOKKOS_LAMBDA`: Captures variables by value and enables GPU execution spaces.

# Multi-Node Computing: Distributed Memory with MPI

**Message Passing Interface (MPI)**

- **Distributed Architecture:** Connects independent compute nodes with private address spaces over high-speed networks.
- **Process Ranks:** Assigns unique ID numbers (`MPI_Comm_rank`) to each running process.
- **Point-to-Point:** Direct process-to-process message transfers via `MPI_Send` and `MPI_Recv`.
- **Collectives:** Synchronized operations across process groups via `MPI_Bcast` and `MPI_Allreduce`.




## Exascale Scalability

MPI provides the foundation for scaling scientific software across thousands of distributed supercomputing nodes.

# Parallel Performance Limits: Amdahl's vs Gustafson's Law

## Amdahl's Law (Strong Scaling)

Fixed Problem Size:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

- Serial fraction $1 - f$ bounds maximum speedup.
- If 5% of code is strictly serial ($1 - f = 0.05$), speedup asymptotically caps at **20x** regardless of core count p.

## Gustafson's Law (Weak Scaling)

Scaled Problem Size:

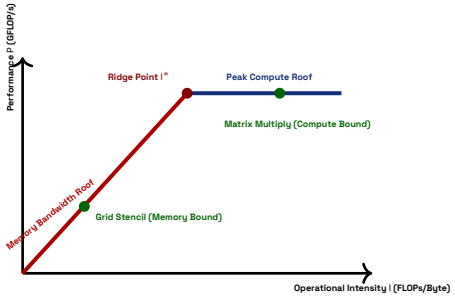
$$S(p) = p - \alpha(p-1)$$

- Workload grows proportionally with processor count p.
- Reflects Exascale HPC practice: Larger clusters enable simulating finer spatial resolutions.

Key Takeaway: Amdahl evaluates time reduction for a fixed workload; Gustafson evaluates problem size expansion in fixed time.

# The Roofline Model: Memory vs Compute Bottlenecks

The **Roofline Model** diagnoses whether application performance is bound by **DRAM Bandwidth** or **Peak Floating-Point Operations**.




**Operational Intensity (I):**

$$I = \frac{\text{Floating Point Operations (FLOPs)}}{\text{DRAM Bytes Transferred}}$$

## Optimization Rule

If code is Memory-Bound ($I < I^*$), increasing arithmetic throughput will not improve speed: optimize cache reuse and memory access patterns.

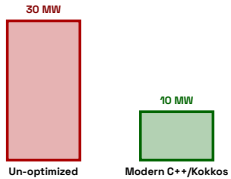
# Green HPC: Problem & Solution in Energy Efficiency

## The Energy Problem

- Top-tier supercomputers draw **20 to 30 MegaWatts** of electrical power, creating high operational expenditure and carbon footprint.
- Inefficient software wastes compute cycles and power budgets.

## Software  Hardware Solutions

- Accelerators (GPUs) offer higher throughput per Watt than general-purpose CPU cores.
- Optimizing code efficiency improves Joules/FLOP and reduces overall execution energy.




### 3x Energy Reduction

Software optimization directly reduces megawatt-hour consumption and operational carbon impact!

# Software Engineering Tools for High-Performance Code

### Modern CMake Build Setup

```
cmake_minimum_required(VERSION 3.21)
project(HPC_App LANGUAGES CXX)

find_package(MPI REQUIRED)
find_package(Kokkos REQUIRED)

add_executable(app src/main.cpp)
target_link_libraries(app PRIVATE
  Kokkos::kokkos
  MPI::MPI_CXX
)
```

### SLURM Job Submission Script

```
#!/bin/bash
#SBATCH --job-name=hpc_sim
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=32
#SBATCH --cpus-per-task=4
#SBATCH --time=02:00:00

module load gcc/12 openmpi/4.1 kokkos/4.2
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun ./app --input input.yaml
```

## Automated CTest Suites

Integrating automated testing (CTest) verifies numerical correctness and regression prevention after code optimizations.

# Recap: Essential Takeaways

Summary of HPC core concepts and practical methods:

**Core Concepts**

- ✓ **Parallel Hardware:** Multi-core CPUs, GPU Accelerators, and Clusters
- ✓ **Thread Safety:** Protecting memory with Atomics and Mutexes
- ✓ **Performance Portability:** Single C++ source abstraction with Kokkos

**Practical Skills**

- ✓ **Multi-Node Scaling:** Distributed computing with MPI
- ✓ **Roofline Analysis:** Diagnosing memory vs compute bottlenecks
- ✓ **Green Computing:** Energy-conscious high-throughput software

**High-performance, portable scientific software engineering!**

# Questions & Discussion

## High-Performance Computing Research


**AHMED EZ-ZOUINE**

**Email:** ahmed.ezzouine@um5r.ac.ma