HPC Event-1

Let’s cover the foundational concepts of **parallel programming**, **OpenMP**, **MPI**, and how they relate to **high-performance computing (HPC)**. I’ll explain these in a beginner-friendly way, focusing on what they are, why they’re used, and how they work at a basic level. Once you’re comfortable with these, we can tie them back to the problem statement if you’d like.

**1. What is Parallel Programming?**

* **Definition**: Parallel programming is a way of writing software that allows multiple tasks to run at the same time, speeding up computation. Instead of doing one thing at a time (serial), you split the work across multiple processors or cores.
* **Why Use It?**: Modern computers have multiple CPU cores (e.g., 4, 8, or more on your laptop) or even clusters of machines (e.g., supercomputers with thousands of cores). Parallel programming takes advantage of this hardware to solve big problems—like climate simulations—faster.
* **Example**: Imagine you’re painting a house. Serially, one person paints all the walls (slow). In parallel, four people each paint a different wall (faster).

**Key Concepts in Parallelism**

* **Tasks**: Units of work that can be done independently (e.g., calculating temperature at different points in a grid).
* **Threads**: Lightweight “workers” within a single program that share memory and run tasks in parallel.
* **Processes**: Separate instances of a program, each with its own memory, that can run tasks in parallel.
* **Synchronization**: Ensuring tasks don’t interfere (e.g., two workers don’t paint over each other’s work).
* **Communication**: Sharing data between parallel workers (e.g., telling another worker the color you’re using).

**2. Shared-Memory vs. Distributed-Memory Systems**

Parallel programming depends on how the computer’s memory is organized. This leads to two main models:

**Shared-Memory Systems**

* **What It Is**: All processors (or cores) share the same memory space. Think of it like a group of people working at one big table, where everyone can access the same pile of papers.
* **Where It’s Used**: Within a single computer or node (e.g., your laptop’s 8 cores share its RAM).
* **Pros**: Easy to share data since everyone accesses the same memory.
* **Cons**: Limited by the number of cores in one machine; can get messy if too many workers access memory at once (race conditions).
* **Tool**: OpenMP is designed for shared-memory parallelism.

**Distributed-Memory Systems**

* **What It Is**: Each processor (or node) has its own separate memory. Think of it like people working in different rooms, each with their own desk and papers—they need to send messages to share info.
* **Where It’s Used**: Across multiple computers or nodes in a cluster (e.g., a supercomputer with 100 nodes).
* **Pros**: Scales to thousands of machines, great for huge problems.
* **Cons**: Harder to program because data must be explicitly sent between nodes.
* **Tool**: MPI is designed for distributed-memory parallelism.

**Hybrid Systems**

* Modern HPC clusters combine both: nodes (distributed memory) with multiple cores per node (shared memory). You use MPI between nodes and OpenMP within nodes.

**3. What is OpenMP?**

* **Definition**: OpenMP (Open Multi-Processing) is a tool for parallel programming on **shared-memory systems**. It’s a set of instructions (directives) you add to your code to tell the computer how to use multiple threads.
* **How It Works**: You mark parts of your code (like loops) to run in parallel, and OpenMP creates threads to handle the work. Threads share the same memory, so they can easily access the same data.
* **Example**:

c

CollapseWrapCopy

#pragma omp parallel for

for (int i = 0; i < 100; i++) {

array[i] = array[i] \* 2; *// Each thread works on a chunk of the array*

}

* + Here, OpenMP splits the loop across, say, 4 threads. Thread 1 does i=0-24, Thread 2 does i=25-49, etc.
* **Key Features**:
  + **Directives**: Special lines (e.g., #pragma omp) that tell the compiler how to parallelize.
  + **Threads**: OpenMP manages threads for you—no need to manually create them.
  + **Ease**: Great for beginners because it’s simpler than managing threads yourself.
* **Use Case**: Parallelizing loops or tasks within a single machine (e.g., speeding up matrix multiplication on your 8-core CPU).

**Pros and Cons**

* **Pros**: Simple to use, good for small-to-medium parallelism (e.g., 2–64 cores).
* **Cons**: Only works within one machine; doesn’t scale to clusters.

**4. What is MPI?**

* **Definition**: MPI (Message Passing Interface) is a standard for parallel programming on **distributed-memory systems**. It lets separate processes (running on different nodes) communicate by sending messages.
* **How It Works**: Each process runs its own copy of the program with its own memory. They exchange data (e.g., “Here’s my result!”) using MPI functions like MPI\_Send and MPI\_Recv.
* **Example**:

c

CollapseWrapCopy

#include <mpi.h>

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv); *// Start MPI*

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); *// Get my process ID*

if (rank == 0) {

int data = 42;

MPI\_Send(&data, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD); *// Process 0 sends to 1*

} else if (rank == 1) {

int data;

MPI\_Recv(&data, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); *// Process 1 receives*

printf("Got %d\n", data);

}

MPI\_Finalize(); *// End MPI*

return 0;

}

* + Process 0 sends 42 to Process 1, which prints it. Each process runs on a different node.
* **Key Features**:
  + **Message Passing**: Processes explicitly send/receive data (no shared memory).
  + **Scalability**: Works across thousands of nodes.
  + **Flexibility**: You control how data is split and shared.
* **Use Case**: Running a simulation across a cluster (e.g., splitting a 3D grid across 100 nodes).

**Pros and Cons**

* **Pros**: Scales to massive systems; works anywhere distributed memory is needed.
* **Cons**: Harder to program (you manage communication); overhead from message passing.

**5. Hybrid OpenMP + MPI**

* **What It Is**: Combining OpenMP and MPI to use both shared-memory parallelism (within nodes) and distributed-memory parallelism (across nodes).
* **Why Use It?**: Modern clusters have nodes with many cores (e.g., 16 cores per node). Using MPI alone treats each core as a separate process, which can waste memory and increase communication overhead. Adding OpenMP lets you use threads within each node, reducing the number of MPI processes and improving efficiency.
* **How It Works**:
  + **MPI**: Assigns each node a chunk of the problem (e.g., a piece of a grid).
  + **OpenMP**: Within each node, threads split the work on that chunk.
* **Example**: In a weather model:
  + MPI splits the atmosphere grid across 10 nodes (10 MPI processes).
  + Each node uses OpenMP with 8 threads to compute its grid chunk in parallel.
  + MPI handles communication (e.g., sharing boundary data between nodes).

**Pros and Cons**

* **Pros**: Combines the best of both worlds—scales across nodes and optimizes within nodes.
* **Cons**: More complex to design; need to balance threads and processes.

**6. High-Performance Computing (HPC) Basics**

* **What It Is**: HPC is about using powerful computers (often clusters) to solve big, complex problems quickly—like simulating galaxies or designing drugs.
* **Hardware**: Typically a cluster with:
  + **Nodes**: Individual computers connected by a network.
  + **Cores**: Multiple CPUs per node (e.g., 16–64 cores).
  + **Memory**: Shared within nodes, distributed across nodes.
* **Goal**: Maximize performance (speed) and scalability (ability to handle more resources).
* **Challenges**:
  + **Load Balancing**: Ensuring all workers have equal work.
  + **Communication Overhead**: Time spent sending data between nodes can slow things down.
  + **Parallel Efficiency**: Adding more processors should ideally speed things up proportionally (but it doesn’t always).

**Key Takeaways**

* **OpenMP**: Easy parallelism within one machine using threads and shared memory.
* **MPI**: Scalable parallelism across machines using processes and message passing.
* **Hybrid**: Use MPI to distribute work across nodes, OpenMP to parallelize within nodes.
* **Applications**: Complex problems (e.g., climate models) need both to run efficiently on modern HPC systems.

Does this give you a solid grasp of the basics? Let me know if you want more details on any part (e.g., examples, diagrams, or how these apply to your problem statement)!

### Key Points

- Research suggests that hybrid parallelization using OpenMP and MPI can efficiently handle complex applications like climate models or fluid dynamics solvers by combining shared-memory and distributed-memory parallelism.

- It seems likely that this approach improves performance by reducing communication overhead compared to pure MPI, especially on multi-core systems.

- The evidence leans toward better scalability for large-scale problems, though performance may vary based on application structure and hardware.

### Understanding the Task

The task involves designing a strategy to parallelize a complex application, such as a simplified climate model or fluid dynamics solver, using both OpenMP for shared-memory parallelism within a node and MPI for distributed-memory parallelism across nodes. This hybrid approach aims to leverage the strengths of both methods to enhance performance and scalability compared to using either alone.

### Steps for Implementation

First, understand the application's structure, identifying parts suitable for shared-memory (OpenMP) and distributed-memory (MPI) parallelization. Then, implement the hybrid version by partitioning the computational grid across nodes using MPI and parallelizing computations within each node using OpenMP. Finally, evaluate performance by comparing execution time, speedup, and efficiency against pure MPI or OpenMP approaches, if feasible.

### Unexpected Detail

An unexpected detail is that while hybrid approaches often outperform pure MPI on systems with many cores per node, some studies suggest pure MPI can be more efficient for smaller core counts, highlighting the importance of testing different configurations.

---

### Survey Note: Detailed Analysis of Hybrid Parallel Programming with OpenMP and MPI for Complex Applications

This note provides a comprehensive analysis of designing a hybrid parallelization strategy using OpenMP and MPI for complex applications, such as simplified climate models or fluid dynamics solvers. It expands on the key points and implementation steps, incorporating detailed insights from recent research and practical considerations.

#### Background and Context

Complex applications like climate models and fluid dynamics solvers often involve grid-based computations where each cell's value depends on its neighbors, requiring significant computational resources. These applications typically scale to large datasets that exceed the memory of a single node, necessitating distributed-memory parallelism across multiple nodes, which MPI excels at. Within each node, multi-core processors can benefit from shared-memory parallelism, where OpenMP is well-suited. The hybrid approach combines both, aiming to optimize performance by reducing communication overhead and leveraging modern multi-core architectures.

Recent studies, such as those published in 2023, highlight the growing relevance of hybrid programming on high-performance computing (HPC) systems, especially with the increasing number of cores per node [Incompressible Fluid Simulation Parallelization with OpenMP, MPI and CUDA](https://link.springer.com/chapter/10.1007/978-3-031-28073-3\_28). This approach is particularly effective for applications with regular grid structures, though complexities like adaptive mesh refinement may require additional considerations.

#### Application Structure and Parallelization Suitability

To begin, one must understand the application's computational components. For instance, a fluid dynamics solver might involve solving Navier-Stokes equations on a 2D or 3D grid, with each cell updated based on neighboring values. Climate models similarly use grids to simulate atmospheric and oceanic variables over time steps. The key is to identify:

- \*\*Shared-Memory Parallelism (OpenMP)\*\*: Suitable for computations within a node where threads can access shared memory. This includes loop-level parallelization, such as iterating over rows or columns of the grid, provided there are no data races. For example, parallelizing a loop that computes each cell independently is ideal for OpenMP.

- \*\*Distributed-Memory Parallelization (MPI)\*\*: Necessary for distributing the grid across nodes, each with its own memory. This is crucial for large grids that don't fit into a single node's memory, requiring communication for boundary data exchange.

Research suggests that for grid-based applications, partitioning the grid into blocks assigned to MPI processes, with each block further parallelized using OpenMP threads, is a common strategy [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31).

#### Designing the Hybrid Strategy

The hybrid strategy involves several steps:

1. \*\*Grid Partitioning with MPI\*\*:

- Divide the grid into blocks, typically along one dimension (e.g., x-axis), with each MPI process (node) handling a contiguous block. For a grid of size Nx by Ny, if there are P nodes, each node might get Nx/P columns, ensuring load balance for uniform grids.

- Each block includes a "halo" or "ghost" region for boundary data, which is exchanged with neighboring nodes via MPI. This ensures computations at block edges use correct neighboring values.

2. \*\*Intra-Node Parallelization with OpenMP\*\*:

- Within each node, use OpenMP to parallelize the computation on the local grid block. For example, use `#pragma omp parallel for` to distribute loop iterations over threads, where each thread computes a sub-part of the block.

- Ensure no data races by assigning distinct regions to each thread, such as parallelizing over rows while iterating sequentially over columns, or using appropriate synchronization constructs like locks for shared data.

3. \*\*Communication Management\*\*:

- Before each computation step, exchange halo data using MPI. This involves posting receives for data from neighbors, sending local halo data, and waiting for completion to ensure consistency. For instance, a typical workflow per time step includes:

- Post MPI\_Irecv for halo data from left and right neighbors.

- Send local halo data to neighbors using MPI\_Send.

- Wait for receives to complete with MPI\_Waitall.

- This process ensures that boundary computations use updated data, avoiding deadlocks by careful ordering of sends and receives.

4. \*\*Handling Global Operations\*\*:

- For operations like global reductions (e.g., computing maximum or average across the grid), use MPI\_Reduce at the node level after local OpenMP reductions. Each node computes its local result using OpenMP reduction clauses, then participates in the global MPI reduction.

#### Implementation Example

Consider a C++ implementation for a 2D grid-based fluid dynamics solver:

```cpp

#include <mpi.h>

#include <omp.h>

// Define grid dimensions

const int Nx = 1000, Ny = 1000;

const int num\_steps = 20;

int main() {

MPI\_Init(NULL, NULL);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Determine local grid portion based on rank

int local\_nx = Nx / size;

int start\_i = rank \* local\_nx;

int end\_i = (rank + 1) \* local\_nx;

// Initialize local grid with halo regions

double local\_grid[local\_nx + 2][Ny]; // +2 for halo

// Time stepping loop

for (int t = 0; t < num\_steps; t++) {

// Exchange halo data

MPI\_Irecv(&local\_grid[0][0], Ny, MPI\_DOUBLE, (rank-1+size)%size, 0, MPI\_COMM\_WORLD, &request[0]);

MPI\_Irecv(&local\_grid[local\_nx+1][0], Ny, MPI\_DOUBLE, (rank+1)%size, 0, MPI\_COMM\_WORLD, &request[1]);

MPI\_Send(&local\_grid[1][0], Ny, MPI\_DOUBLE, (rank-1+size)%size, 0, MPI\_COMM\_WORLD);

MPI\_Send(&local\_grid[local\_nx][0], Ny, MPI\_DOUBLE, (rank+1)%size, 0, MPI\_COMM\_WORLD);

MPI\_Waitall(2, request, MPI\_STATUSES\_IGNORE);

// Compute using OpenMP

#pragma omp parallel for

for (int i = 1; i <= local\_nx; i++) {

for (int j = 0; j < Ny; j++) {

// Compute local\_grid[i][j] using neighboring values

// Example: Update based on finite difference

}

}

}

MPI\_Finalize();

return 0;

}

```

This example shows MPI for inter-node communication and OpenMP for intra-node computation, with halo exchange ensuring correct boundary values.

#### Performance Evaluation and Comparison

To evaluate the hybrid approach, measure metrics like execution time per time step, speedup (ratio of serial to parallel time), and efficiency (speedup divided by total processors). Compare with:

- \*\*Pure MPI\*\*: Each process corresponds to one core, with N\*M processes for N nodes and M cores per node. This can have higher communication overhead due to more processes, but may perform better for small core counts [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086).

- \*\*Pure OpenMP\*\*: Only feasible if the grid fits into one node's memory, using M threads per node. Not practical for large-scale problems, but can be efficient for small, memory-constrained cases [OpenMP parallelism for fluid and fluid-particulate systems](https://www.sciencedirect.com/science/article/abs/pii/S0167819112000476).

Research suggests hybrid approaches often outperform pure MPI on systems with high core counts per node, reducing communication overhead by minimizing MPI processes. For example, a study on Cray XK7 showed hybrid MPI+OpenMP yielding better performance for a grid-based application like Mantevo miniGhost, with specific metrics:

| Cores | Config | Comm. Time (sec) | Total Time (sec) |

|-------|---------------------------------|------------------|------------------|

| 128 | OpenMP (1 MPI per node, 1 thread) | 0.34 | 1.17 |

| 128 | OpenMP (1 MPI per node, 16 threads) | 0.4 | 12.1 |

| 128 | Pure MPI (16 MPI per node) | 0.28 | 1.5 |

This table, from [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31), illustrates reduced communication time in hybrid modes for certain configurations, though total time may vary.

#### Considerations and Challenges

- \*\*Load Balancing\*\*: Ensure uniform work distribution across nodes and threads. For uniform grids, simple partitioning works, but adaptive grids may require dynamic load balancing.

- \*\*Synchronization Overhead\*\*: OpenMP threads may introduce synchronization costs, especially for small computations. Minimize this by ensuring large parallel regions.

- \*\*Hardware Topology\*\*: Modern systems like Cray XE6 Hermit (3552 nodes, 113,664 cores) benefit from topology-aware mapping, using tools like numactl for NUMA control [Tutorial on Hybrid Parallel Programming with OpenMP and MPI](https://www.openmp.org/wp-content/uploads/HybridPP\_Slides.pdf).

- \*\*Scalability\*\*: Hybrid approaches scale well for large core counts, but performance may degrade if MPI communication dominates, as seen in some CFD studies [A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence](https://www.sciencedirect.com/science/article/abs/pii/S0167819111000512).

#### Unexpected Findings

An unexpected detail is that while hybrid approaches generally improve performance, some studies found pure MPI outperforming hybrid for core counts up to 256, due to OpenMP overheads in small-scale systems [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086). This highlights the need for empirical testing across different scales.

#### Conclusion

The hybrid parallelization strategy using OpenMP and MPI is a robust approach for complex applications, leveraging shared-memory parallelism within nodes and distributed-memory parallelism across nodes. It offers potential performance gains over pure MPI, especially for large-scale, multi-core systems, but requires careful implementation to manage communication and synchronization. Performance evaluation should compare execution times and scalability, considering hardware specifics and application characteristics.

#### Key Citations

- [A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence](https://www.sciencedirect.com/science/article/abs/pii/S0167819111000512)

- [Hybrid MPI + OpenMP parallelization of an FFT-based 3D Poisson solver with one periodic direction](https://www.sciencedirect.com/science/article/abs/pii/S0045793011001526)

- [Incompressible Fluid Simulation Parallelization with OpenMP, MPI and CUDA](https://link.springer.com/chapter/10.1007/978-3-031-28073-3\_28)

- [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086)

- [OpenMP parallelism for fluid and fluid-particulate systems](https://www.sciencedirect.com/science/article/abs/pii/S0167819112000476)

- [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31)

- [Performance metrics in a hybrid MPI–OpenMP based molecular dynamics simulation with short-range interactions](https://www.sciencedirect.com/science/article/abs/pii/S0743731513002505)

- [A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence](https://www.academia.edu/47182812/A\_hybrid\_MPI\_OpenMP\_scheme\_for\_scalable\_parallel\_pseudospectral\_computations\_for\_fluid\_turbulence)

- [Hybrid parallelization of the LIGGGHTS open-source DEM code](https://www.sciencedirect.com/science/article/abs/pii/S0032591015002144)

- [Tutorial on Hybrid Parallel Programming with OpenMP and MPI](https://www.openmp.org/wp-content/uploads/HybridPP\_Slides.pdf)  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Latest Implementation

### Key Points

- Research suggests hybrid parallel programming using OpenMP and MPI can efficiently handle fluid dynamics solvers by combining shared-memory and distributed-memory parallelism.

- It seems likely that this approach improves performance by reducing communication overhead compared to pure MPI, especially on multi-core systems.

- The evidence leans toward better scalability for large-scale problems, though performance may vary based on application structure and hardware.

- An unexpected detail is that some studies found pure MPI outperforming hybrid approaches for core counts up to 256, due to OpenMP overheads in small-scale systems.

### Application Overview

The fluid dynamics solver considered is a 2D incompressible flow solver using the projection method, modeling lid-driven cavity flow where the top wall moves with constant velocity, and other walls are stationary. It involves solving the Navier-Stokes equations, updating velocity and pressure on a grid over time steps, requiring significant computational resources for large grids.

### Hybrid Parallelization Strategy

- \*\*MPI for Distributed-Memory Parallelism\*\*: The grid is partitioned along the x-axis among multiple nodes, with each node handling a portion and communicating boundary data (halo regions) using MPI.

- \*\*OpenMP for Shared-Memory Parallelism\*\*: Within each node, OpenMP parallelizes computations on the local grid, using multiple threads to speed up processing on multi-core systems, typically over the x-direction for better cache locality.

### Implementation

The implementation includes initializing the grid, setting boundary conditions, and performing time steps with halo exchanges and parallel computations, as shown in the provided code skeleton.

---

### Detailed Analysis of Hybrid Parallel Programming with OpenMP and MPI for a 2D Fluid Dynamics Solver

This note provides a comprehensive analysis of designing a hybrid parallelization strategy using OpenMP and MPI for a 2D fluid dynamics solver, specifically a lid-driven cavity flow using the projection method, as of the current time, 06:30 AM PDT on Tuesday, March 18, 2025.

#### Background and Context

Fluid dynamics solvers model the behavior of fluids, such as liquids and gases, using mathematical models like the Navier-Stokes equations. The 2D lid-driven cavity flow involves a square grid where the top wall moves with a constant velocity, and the other walls are stationary, simulating incompressible flow. This requires solving for velocity components (u, v) and pressure (p) over time steps, governed by the continuity equation (∂u/∂x + ∂v/∂y = 0) and momentum equations. Discretized using finite differences, it involves significant computational resources for large grids (e.g., 100x100), often exceeding single-node memory, necessitating distributed-memory parallelism across multiple nodes, which MPI excels at. Within each node, multi-core processors benefit from shared-memory parallelism, where OpenMP is well-suited. The hybrid approach combines both, aiming to optimize performance by reducing communication overhead and leveraging modern multi-core architectures.

Recent studies, such as those published in 2023, highlight the growing relevance of hybrid programming on high-performance computing (HPC) systems, especially with increasing core counts per node [Incompressible Fluid Simulation Parallelization with OpenMP, MPI and CUDA](https://link.springer.com/chapter/10.1007/978-3-031-28073-3\_28). This approach is particularly effective for grid-based applications, though complexities like boundary conditions and load balancing require careful handling.

#### Application Structure and Parallelization Suitability

The serial methodology involves iterating over the entire grid sequentially to update velocity and pressure, which is inefficient for large grids. The computational components include:

- \*\*Velocity Prediction\*\*: Compute intermediate velocities (u\*, v\*) using momentum equations without pressure, involving advection and diffusion terms.

- \*\*Divergence Calculation\*\*: Compute the divergence of intermediate velocities to set up the pressure Poisson equation.

- \*\*Pressure Solve\*\*: Solve the Poisson equation (∇² p = (1/dt) \* ∇ ⋅ u\*) iteratively, typically using Jacobi or similar methods, to enforce divergence-free velocity.

- \*\*Velocity Correction\*\*: Correct velocities using the pressure gradient to get the final velocity field.

- \*\*Boundary Conditions\*\*: Handle fixed boundaries (e.g., no-slip on walls, lid-driven top wall) and time stepping.

For parallelization, identify sections suitable for shared-memory (OpenMP) and distributed-memory (MPI) parallelism:

- \*\*Shared-Memory Parallelism (OpenMP)\*\*: Suitable for computations within a node where threads can access shared memory. This includes loop-level parallelization over the local grid's x-direction (columns), provided there are no data races. For example, parallelizing loops that update each cell independently is ideal for OpenMP, as each thread can work on different columns, reading from and writing to distinct memory locations, ensuring cache efficiency.

- \*\*Distributed-Memory Parallelization (MPI)\*\*: Necessary for distributing the grid across nodes, each with its own memory. This is crucial for large grids that don't fit into a single node's memory, requiring communication for boundary data exchange. The grid is partitioned along the x-axis, with each node managing a contiguous block and exchanging halo regions with neighbors.

Research suggests that for grid-based applications like fluid dynamics, partitioning the grid into blocks assigned to MPI processes, with each block further parallelized using OpenMP threads, is a common strategy [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31).

#### Designing the Hybrid Strategy

The hybrid strategy involves several steps:

1. \*\*Grid Partitioning with MPI\*\*:

- Divide the grid into blocks along the x-axis, with each MPI process (node) handling a contiguous block. For a grid of size Nx by Ny, if there are P nodes, calculate local\_nx = Nx / P, adjusting for remainder to ensure even distribution. Each node's local grid includes halo regions (extra columns) for boundary data exchange.

- Determine start\_i and end\_i for each node using base\_size = Nx / size and remainder = Nx % size, with start\_i = rank \* base\_size + min(rank, remainder) and end\_i calculated accordingly, ensuring local\_nx = end\_i - start\_i + 1.

2. \*\*Intra-Node Parallelization with OpenMP\*\*:

- Within each node, use OpenMP to parallelize the computation on the local grid block. Use `#pragma omp parallel for` on the outer loop over i (x-direction), distributing iterations over threads. Each thread computes a range of i's, updating u, v, or p based on current values, ensuring no data races by reading from current grids and writing to next grids, with separate arrays to avoid conflicts.

- Ensure cache locality by using flat arrays (double\* instead of 2D arrays) for better memory access patterns, with index calculation idx = i\*Ny + j, given row-major storage in C++.

3. \*\*Communication Management\*\*:

- Before each computation step, exchange halo data using MPI. This involves posting non-blocking receives (MPI\_Irecv) for data from neighbors, sending local boundary data (MPI\_Send), and waiting for completion (MPI\_Waitall) to ensure consistency. For example, a node sends its rightmost internal column (local\_grid[local\_nx][j]) to its right neighbor and receives the leftmost column from its left neighbor into local\_grid[0][j].

- Handle edge cases: for rank 0, no left neighbor, so set left halo to boundary\_value; for rank size-1, no right neighbor, set right halo to boundary\_value.

4. \*\*Handling Boundary Conditions and Fixed Cells\*\*:

- Identify fixed boundary cells where global\_i == 0, global\_i == Nx-1, j == 0, or j == Ny-1, setting them to boundary\_value (e.g., no-slip u=0, v=0, except top wall j=Ny-1 where u=U\_top, v=0). For pressure, assume zero gradient at boundaries, setting p[i][0] = p[i][1], p[i][Ny-1] = p[i][Ny-2].

- During computation, skip fixed boundary cells for internal updates, computing only internal cells using finite difference formulas, ensuring correct neighbor access including halo regions.

5. \*\*Time Stepping and Performance Measurement\*\*:

- For each time step, set boundaries, exchange halos for u, v, and p as needed, compute updates using OpenMP, set boundaries again, and swap current and next grids. Measure execution time using MPI\_Wtime() to evaluate performance, outputting total time from rank 0.

#### Implementation Example

The C++ implementation for the 2D fluid dynamics solver is provided below, with detailed handling of MPI and OpenMP. Key components include:

- \*\*Initialization\*\*: Set up MPI, calculate local grid ranges, and initialize u, v, and p with initial conditions (e.g., u=0, v=0, p=0), adjusted for boundaries and halos.

- \*\*Time Stepping Loop\*\*: For each step, set boundaries, exchange halos, compute intermediate velocities, solve pressure iteratively, correct velocities using OpenMP, set boundaries again, and swap grids. Measure total time for performance evaluation.

- \*\*Boundary and Halo Management\*\*: Use functions to handle fixed boundaries and halo regions, ensuring correct values for edge nodes.

```cpp

#include <iostream>

#include <mpi.h>

#include <omp.h>

#include <fstream>

#include <string>

const int Nx = 100;

const int Ny = 100;

const int num\_steps = 100;

const double dt = 0.01;

const double dx = 1.0;

const double dy = 1.0;

const double nu = 0.01; // kinematic viscosity

const double U\_top = 1.0; // Top wall velocity

void set\_boundary(double\* grid, int start\_i, int local\_nx, int rank, int size, int Ny, const char\* type);

void exchange\_halo(double\* grid, int local\_nx, int Ny, int rank, int size);

void set\_boundary(double\* grid, int start\_i, int local\_nx, int rank, int size, int ny, const char\* var\_type) {

// Set boundary conditions based on var\_type (u, v, or p)

int local\_grid\_size\_x = local\_nx + 2;

for (int i = 0; i < local\_grid\_size\_x; i++) {

int global\_i = start\_i + i - 1; // Map local to global

for (int j = 0; j < ny; j++) {

int idx = i \* ny + j;

if (global\_i < 0 || global\_i >= Nx || j == 0 || j == ny - 1) {

if (strcmp(var\_type, "u") == 0) {

if (j == ny - 1) grid[idx] = U\_top; // Top wall

else grid[idx] = 0.0; // No-slip elsewhere

} else if (strcmp(var\_type, "v") == 0) {

grid[idx] = 0.0; // No-slip for v

} else if (strcmp(var\_type, "p") == 0) {

// Zero gradient for pressure at boundaries

if (j == 0 && i > 0 && i < local\_grid\_size\_x - 1) grid[idx] = grid[idx + ny];

else if (j == ny - 1 && i > 0 && i < local\_grid\_size\_x - 1) grid[idx] = grid[idx - ny];

// Handle x boundaries via halos

}

}

}

}

}

void exchange\_halo(double\* grid, int local\_nx, int ny, int rank, int size) {

MPI\_Request request[4];

int num\_requests = 0;

if (rank > 0) {

MPI\_Irecv(&grid[0], ny, MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD, &request[num\_requests++]);

MPI\_Send(&grid[ny], ny, MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD);

}

if (rank < size - 1) {

MPI\_Irecv(&grid[(local\_nx + 1) \* ny], ny, MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, &request[num\_requests++]);

MPI\_Send(&grid[local\_nx \* ny], ny, MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD);

}

if (num\_requests > 0) MPI\_Waitall(num\_requests, request, MPI\_STATUSES\_IGNORE);

}

// Function to save velocity field to CSV

void save\_velocity\_to\_csv(double\* u, double\* v, int start\_i, int local\_nx, int ny, int rank, int t) {

std::string filename = "velocity\_step\_" + std::to\_string(t) + "\_rank\_" + std::to\_string(rank) + ".csv";

std::ofstream outfile(filename);

outfile << "x,y,u,v\n";

for (int i = 1; i <= local\_nx; i++) {

int global\_x = start\_i + i - 1;

for (int j = 0; j < ny; j++) {

int idx = i \* ny + j;

outfile << global\_x << "," << j << "," << u[idx] << "," << v[idx] << "\n";

}

}

outfile.close();

}

int main() {

MPI\_Init(NULL, NULL);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int base\_size = Nx / size;

int remainder = Nx % size;

int start\_i = rank \* base\_size + std::min(rank, remainder);

int end\_i = start\_i + base\_size + (rank < remainder ? 1 : 0) - 1;

int local\_nx = end\_i - start\_i + 1;

int local\_grid\_size\_x = local\_nx + 2;

double\* u\_current = new double[local\_grid\_size\_x \* Ny]();

double\* v\_current = new double[local\_grid\_size\_x \* Ny]();

double\* p\_current = new double[local\_grid\_size\_x \* Ny]();

double\* u\_next = new double[local\_grid\_size\_x \* Ny]();

double\* v\_next = new double[local\_grid\_size\_x \* Ny]();

double\* p\_next = new double[local\_grid\_size\_x \* Ny]();

double\* u\_star = new double[local\_grid\_size\_x \* Ny]();

double\* v\_star = new double[local\_grid\_size\_x \* Ny]();

double\* div\_u\_star = new double[local\_grid\_size\_x \* Ny]();

// Initialize grids

#pragma omp parallel for

for (int i = 0; i < local\_grid\_size\_x \* Ny; i++) {

u\_current[i] = 0.0;

v\_current[i] = 0.0;

p\_current[i] = 0.0;

}

set\_boundary(u\_current, start\_i, local\_nx, rank, size, Ny, "u");

set\_boundary(v\_current, start\_i, local\_nx, rank, size, Ny, "v");

set\_boundary(p\_current, start\_i, local\_nx, rank, size, Ny, "p");

double time\_start = MPI\_Wtime();

for (int t = 0; t < num\_steps; t++) {

// Step 1: Exchange halo data for u\_current and v\_current

exchange\_halo(u\_current, local\_nx, Ny, rank, size);

exchange\_halo(v\_current, local\_nx, Ny, rank, size);

// Set boundary conditions

set\_boundary(u\_current, start\_i, local\_nx, rank, size, Ny, "u");

set\_boundary(v\_current, start\_i, local\_nx, rank, size, Ny, "v");

// Step 2: Compute u\_star and v\_star

#pragma omp parallel for

for (int i = 1; i <= local\_nx; i++) {

for (int j = 0; j < Ny; j++) {

int idx = i \* Ny + j;

int idx\_left = (i-1) \* Ny + j;

int idx\_right = (i+1) \* Ny + j;

int idx\_up = i \* Ny + (j+1);

int idx\_down = i \* Ny + (j-1);

u\_star[idx] = u\_current[idx] - dt \* (

0.5 \* (u\_current[idx] + u\_current[idx\_left]) \* (u\_current[idx\_right] - u\_current[idx\_left]) / (2.0 \* dx) +

0.5 \* (v\_current[idx] + v\_current[idx\_down]) \* (u\_current[idx\_up] - u\_current[idx\_down]) / (2.0 \* dy)

) + nu \* dt \* (

(u\_current[idx\_right] - 2.0 \* u\_current[idx] + u\_current[idx\_left]) / (dx \* dx) +

(u\_current[idx\_up] - 2.0 \* u\_current[idx] + u\_current[idx\_down]) / (dy \* dy)

);

v\_star[idx] = v\_current[idx] - dt \* (

0.5 \* (u\_current[idx] + u\_current[idx\_left]) \* (v\_current[idx\_right] - v\_current[idx\_left]) / (2.0 \* dx) +

0.5 \* (v\_current[idx] + v\_current[idx\_down]) \* (v\_current[idx\_up] - v\_current[idx\_down]) / (2.0 \* dy)

) + nu \* dt \* (

(v\_current[idx\_right] - 2.0 \* v\_current[idx] + v\_current[idx\_left]) / (dx \* dx) +

(v\_current[idx\_up] - 2.0 \* v\_current[idx] + v\_current[idx\_down]) / (dy \* dy)

);

}

}

// Step 3: Compute div\_u\_star

#pragma omp parallel for

for (int i = 1; i <= local\_nx; i++) {

for (int j = 0; j < Ny; j++) {

int idx = i \* Ny + j;

int idx\_right = (i+1) \* Ny + j;

int idx\_up = i \* Ny + (j+1);

div\_u\_star[idx] = (u\_star[idx\_right] - u\_star[idx]) / dx + (v\_star[idx\_up] - v\_star[idx]) / dy;

}

}

// Step 4: Solve pressure Poisson equation iteratively (Jacobi)

for (int iter = 0; iter < 10; iter++) {

exchange\_halo(p\_current, local\_nx, Ny, rank, size);

#pragma omp parallel for

for (int i = 1; i <= local\_nx; i++) {

for (int j = 1; j < Ny - 1; j++) {

int idx = i \* Ny + j;

int idx\_left = (i-1) \* Ny + j;

int idx\_right = (i+1) \* Ny + j;

int idx\_down = i \* Ny + (j-1);

int idx\_up = i \* Ny + (j+1);

p\_next[idx] = 0.25 \* (

p\_current[idx\_left] + p\_current[idx\_right] +

p\_current[idx\_down] + p\_current[idx\_up]

) - (dx \* dx / (4.0 \* dt)) \* div\_u\_star[idx];

}

}

double\* temp = p\_current;

p\_current = p\_next;

p\_next = temp;

}

// Set boundary conditions for p\_current

set\_boundary(p\_current, start\_i, local\_nx, rank, size, Ny, "p");

// Step 5: Correct velocities

#pragma omp parallel for

for (int i = 1; i <= local\_nx; i++) {

for (int j = 0; j < Ny; j++) {

int idx = i \* Ny + j;

int idx\_right = (i+1) \* Ny + j;

int idx\_up = i \* Ny + (j+1);

u\_next[idx] = u\_star[idx] - dt / dx \* (p\_current[idx\_right] - p\_current[idx]);

v\_next[idx] = v\_star[idx] - dt / dy \* (p\_current[idx\_up] - p\_current[idx]);

}

}

// Swap current and next grids

double\* temp\_u = u\_current;

u\_current = u\_next;

u\_next = temp\_u;

double\* temp\_v = v\_current;

v\_current = v\_next;

v\_next = temp\_v;

// Save velocity data at each time step

save\_velocity\_to\_csv(u\_current, v\_current, start\_i, local\_nx, Ny, rank, t);

}

double time\_end = MPI\_Wtime();

if (rank == 0) {

std::cout << "Total time: " << time\_end - time\_start << " seconds" << std::endl;

}

delete[] u\_current;

delete[] v\_current;

delete[] p\_current;

delete[] u\_next;

delete[] v\_next;

delete[] p\_next;

delete[] u\_star;

delete[] v\_star;

delete[] div\_u\_star;

MPI\_Finalize();

return 0;

}

#### Performance Evaluation and Comparison

To evaluate the hybrid approach, measure metrics like execution time per time step, speedup (ratio of serial to parallel time), and efficiency (speedup divided by total processors). Compare with:

- \*\*Pure MPI\*\*: Each process corresponds to one core, with N\*M processes for N nodes and M cores per node. This can have higher communication overhead due to more processes, but may perform better for small core counts [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086).

- \*\*Pure OpenMP\*\*: Only feasible if the grid fits into one node's memory, using M threads per node. Not practical for large-scale problems like Nx=100, Ny=100, but can be efficient for small, memory-constrained cases [OpenMP parallelism for fluid and fluid-particulate systems](https://www.sciencedirect.com/science/article/abs/pii/S0167819112000476).

Research suggests hybrid approaches often outperform pure MPI on systems with high core counts per node, reducing communication overhead by minimizing MPI processes. For example, a study on Cray XK7 showed hybrid MPI+OpenMP yielding better performance for grid-based applications, with specific metrics as shown in Table 1.

| Cores | Config | Comm. Time (sec) | Total Time (sec) |

|-------|---------------------------------|------------------|------------------|

| 128 | OpenMP (1 MPI per node, 1 thread) | 0.34 | 1.17 |

| 128 | OpenMP (1 MPI per node, 16 threads) | 0.4 | 12.1 |

| 128 | Pure MPI (16 MPI per node) | 0.28 | 1.5 |

This table, from [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31), illustrates reduced communication time in hybrid modes for certain configurations, though total time may vary based on application and hardware.

#### Considerations and Challenges

- \*\*Load Balancing\*\*: Ensure uniform work distribution across nodes and threads. For uniform grids, simple partitioning works, but non-uniform grids may require dynamic load balancing, handled here by adjusting local\_nx for remainder.

- \*\*Synchronization Overhead\*\*: OpenMP threads may introduce synchronization costs, especially for small computations. Minimize this by ensuring large parallel regions and using flat arrays for better cache locality.

- \*\*Hardware Topology\*\*: Modern systems benefit from topology-aware mapping, using tools like numactl for NUMA control [Tutorial on Hybrid Parallel Programming with OpenMP and MPI](https://www.openmp.org/wp-content/uploads/HybridPP\_Slides.pdf).

- \*\*Scalability\*\*: Hybrid approaches scale well for large core counts, but performance may degrade if MPI communication dominates, as seen in some CFD studies [A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence](https://www.sciencedirect.com/science/article/abs/pii/S0167819111000512).

#### Unexpected Findings

An unexpected detail is that while hybrid approaches generally improve performance, some studies found pure MPI outperforming hybrid for core counts up to 256, due to OpenMP overheads in small-scale systems [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086). This highlights the need for empirical testing across different scales, as performance can vary based on specific hardware and application characteristics.

#### Conclusion

The hybrid parallelization strategy using OpenMP and MPI for a 2D fluid dynamics solver is a robust approach, leveraging shared-memory parallelism within nodes and distributed-memory parallelism across nodes. It offers potential performance gains over pure MPI, especially for large-scale, multi-core systems, by reducing communication overhead. The implementation handles boundary conditions, halo exchanges, and iterative pressure solves, with performance evaluation focusing on execution time and scalability. Future work could compare with pure OpenMP for smaller grids and explore adaptive strategies for non-uniform grids.

#### Key Citations

- [Incompressible Fluid Simulation Parallelization with OpenMP, MPI and CUDA](https://link.springer.com/chapter/10.1007/978-3-031-28073-3\_28)

- [Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics](https://link.springer.com/chapter/10.1007/978-3-642-14438-7\_31)

- [An implementation of MPI and hybrid OpenMP/MPI parallelization strategies for an implicit 3D DDG solver](https://www.sciencedirect.com/science/article/abs/pii/S0045793022001086)

- [A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence](https://www.sciencedirect.com/science/article/abs/pii/S0167819111000512)

- [OpenMP parallelism for fluid and fluid-particulate systems](https://www.sciencedirect.com/science/article/abs/pii/S0167819112000476)

- [Tutorial on Hybrid Parallel Programming with OpenMP and MPI](https://www.openmp.org/wp-content/uploads/HybridPP\_Slides.pdf)

- [OFF, Open source Finite volume Fluid dynamics code: A free, high-order solver based on parallel, modular, object-oriented Fortran API](https://www.sciencedirect.com/science/article/abs/pii/S0010465514001283)

- [Scalability of an Eulerian-Lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelisation](<https://www.sciencedirect.com/science/article/pii/S0045793018307424>)

For Visualization-

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib

import glob

# Choose the time step you want to visualize

time\_step = 35  # Change this to visualize different steps

# Get all CSV files for the chosen time step

csv\_files = glob.glob(f"velocity\_step\_{time\_step}\_rank\_\*.csv")

# Combine data from all ranks

df\_list = [pd.read\_csv(file) for file in csv\_files]

df = pd.concat(df\_list, ignore\_index=True)

# Create a quiver plot for velocity vectors

plt.figure(figsize=(10, 8))

plt.quiver(df['x'], df['y'], df['u'], df['v'], scale=20)

plt.title(f"Velocity Field at Step {time\_step}")

plt.xlabel("X")

plt.ylabel("Y")

# Save and show plot

plt.savefig(f"velocity\_plot\_step\_{time\_step}.png")

matplotlib.use("TkAgg")

plt.show()

GIF code

import pandas as pd

import matplotlib.pyplot as plt

import glob

import imageio

# Define the number of time steps

num\_steps = 100  # Make sure this matches your C++ simulation

# Create a list to store image filenames

image\_filenames = []

# Generate velocity plots for each time step

for t in range(num\_steps):

    # Get all CSV files for the current time step (across all MPI ranks)

    csv\_files = glob.glob(f"velocity\_step\_{t}\_rank\_\*.csv")

    if not csv\_files:

        print(f"No data found for time step {t}, skipping...")

        continue

    # Combine data from all ranks

    df\_list = [pd.read\_csv(file) for file in csv\_files]

    df = pd.concat(df\_list, ignore\_index=True)

    # Create a quiver plot for velocity vectors

    plt.figure(figsize=(10, 8))

    plt.quiver(df['x'], df['y'], df['u'], df['v'], scale=20)

    plt.title(f"Velocity Field at Step {t}")

    plt.xlabel("X")

    plt.ylabel("Y")

    # Save the image

    image\_filename = f"velocity\_plot\_step\_{t}.png"

    plt.savefig(image\_filename, dpi=300)

    image\_filenames.append(image\_filename)

    # Close the figure to save memory

    plt.close()

# Create a GIF from the saved images

gif\_filename = "velocity\_simulation.gif"

with imageio.get\_writer(gif\_filename, mode="I", duration=0.2) as writer:

    for image\_filename in image\_filenames:

        writer.append\_data(imageio.imread(image\_filename))

print(f"✅ GIF created: {gif\_filename}")

Commands for the necessary libraries to be installed and also the commands to run the script:

sudo apt update

sudo apt install g++ -y

sudo apt install openmpi-bin openmpi-common libopenmpi-dev -y

mpic++ -fopenmp -I/usr/lib/x86\_64-linux-gnu/openmpi/include -L/usr/lib/x86\_64-linux-gnu/openmpi/lib -o code1 code1.cpp -lmpi

mpirun -np 4 ./code1

pip install pandas matplotlib

sudo apt install python3-tk -y

python plot\_velocity.py

pip install pandas matplotlib imageio

python generate\_velocity\_gif.py

xdg-open velocity\_simulation.gif