Assignment 2: 2D Convolution with MPI and OpenMP Hybrid Parallelism

Authors: Jiazheng Guo(24070858),Zichen Zhang(24064091)
Due: Friday, 17th October 2025

1. Introduction

This report aims to provide a detailed description of the parallel implementation and performance analysis of a 2D discrete convolution algorithm. Building upon Assignment 1, this project implements stride-supported convolution operations, achieving distributed memory parallelism via MPI and hybrid parallelism through **OpenMP + MPI**. The report focuses on analyzing the hybrid parallelization strategy, data decomposition, communication mechanisms, and performance metrics and acceleration effects on **Kaya/Setonix** HPC resources.

---

2. Implementation Details

2.1 Serial implementation of 2D convolution with stride

```
/**
 * Serial implementation of 2D convolution with stride and "same" padding
 * Output size: ceil(H/sH) × ceil(W/sW)
 */
void conv2d_serial_stride(float **f, int H, int W, float **g, int kH, int kW, int
sH, int sW, float **output) {
    int pad_top = (kH - 1) / 2;
    int pad_left = (kW - 1) / 2;

    int out_H = (H + sH - 1) / sH;  // ceil(H/sH)
    int out_W = (W + sW - 1) / sW;  // ceil(W/sW)

    // For each output pixel (with stride)
    for (int out_i = 0; out_i < out_H; out_i++) {
        for (int out_j = 0; out_j < out_W; out_j++) {
            float sum = 0.0f;

            // Map output position to input position
            int i = out_i * sH;
            int j = out_j * sW;

            // Convolve with kernel
            for (int ki = 0; ki < kH; ki++) {
                for (int kj = 0; kj < kW; kj++) {
                    int input_i = i + ki - pad_top;
                    int input_j = j + kj - pad_left;

                    if (input_i >= 0 && input_i < H && input_j >= 0 && input_j <
W) {
                        sum += f[input_i][input_j] * g[ki][kj];
                    }
                }
            }
            output[out_i][out_j] = sum;
        }
    }
}
```

The `conv2d_serial` function has been extended with stride parameters `(sH, sW)`. The outer loop iterates over output pixels `(out_i, out_j)`, mapping them to the starting `position (i, j)` within the input array. The output dimensions are calculated as **⌈H/sH⌉ × ⌈W/sW⌉**.

2.2 OpenMP implementation with stride support

```
/**
 * OpenMP implementation with stride support
 */
void conv2d_omp_stride(float **f, int H, int W, float **g, int kH, int kW, int
sH, int sW, float **output) {
             ...

    #pragma omp parallel for schedule(dynamic, 16) collapse(2)
    for (int out_i = 0; out_i < out_H; out_i++) {
        for (int out_j = 0; out_j < out_W; out_j++) {
             ...
            output[out_i][out_j] = sum;
        }
    }
}
```

Parallel version of conv2d_serial_stride. Utilises the `#pragma omp parallel for` directive to parallelise the two outermost loops of the output array (`out_i` , `out_j`). Employs `schedule(dynamic, 16)` and `collapse(2)` for dynamic load balancing and flattening parallelism of the two-dimensional loops, thereby enhancing parallel efficiency.

2.3 MPI-only distributed memory implementation with stride

```
/**
 * MPI-only distributed memory implementation with stride
 *
 * Data decomposition: Row-based decomposition of output
 * Each process computes a contiguous block of output rows
 * Requires halo exchange for overlapping input regions
 */
void conv2d_mpi_stride(float **f, int H, int W, float **g, int kH, int kW, int
sH, int sW, float **output, MPI_Comm comm) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    ...

    // Distribute output rows among processes
    ...

    // Compute local output only if this process has rows assigned
    if (local_rows > 0) {
        // Calculate input region needed (with halo)
        ...

        // Compute local output
```

```
        for (int out_i = 0; out_i < local_rows; out_i++) {
            for (int out_j = 0; out_j < out_W; out_j++) {
                ...
                output[local_start + out_i][out_j] = sum;
            }
        }

        if (size > 1) {
            free_2d_array(local_f, input_rows);
        }
    }

    // Gather results to all processes
    // All processes must participate in all MPI_Bcast calls
    if (size > 1) {
        ...
    }
}
```

Pure MPI implementation. Data decomposition is achieved by distributing output rows to different processes (`rank`). Each process computes its assigned output rows and calculates the required input region (including the **halo**) based on the convolution kernel size and stride. Finally, `MPI_Bcast` is employed to broadcast the results computed by all processes to every process, enabling global synchronisation and result collection.

2.4 Hybrid MPI+OpenMP implementation with stride

```
/**
 * Hybrid MPI+OpenMP implementation with stride
 *
 * Two-level parallelism:
 * - MPI: Distribute output rows across processes
 * - OpenMP: Parallelize computation within each process
 *
 * This is the main function for Assignment 2
 */
void conv2d_stride(float **f, int H, int W, float **g, int kH, int kW, int sH,
int sW, float **output, MPI_Comm comm) {
     ...
    // Distribute output rows among processes
     ...

    // Compute local output only if this process has rows assigned
    if (local_rows > 0) {
        // Calculate input region needed (with halo)
        ...

        // Compute local output with OpenMP parallelization
        #pragma omp parallel for schedule(dynamic, 16) collapse(2)
        for (int out_i = 0; out_i < local_rows; out_i++) {
            for (int out_j = 0; out_j < out_W; out_j++) {
                ...
                output[local_start + out_i][out_j] = sum;
```

```
            }
        }

        if (size > 1) {
            free_2d_array(local_f, input_rows);
        }
    }

    // Gather results to all processes
    // All processes must participate in all MPI_Bcast calls
    if (size > 1) {
       ...
    }
}
```

Combines MPI's distributed memory (inter-process communication) with OpenMP's shared memory (inter-thread parallelism). MPI handles the coarse-grained parallelisation by allocating output rows to processes. Within each MPI process, OpenMP employs parallel for loops (as seen in `conv2d_omp_stride`) to parallelise the locally computed output rows, achieving fine-grained parallelism. Result collection similarly utilises `MPI_Bcast`.

3. Parallelisation Strategy

This project employs a hybrid parallel model combining **OpenMP** and **MPI**, aiming to balance the high latency of inter-process communication in MPI with the low overhead of inter-thread synchronisation in OpenMP.

MPI layer (coarse-grained): Responsible for distributing rows of the output matrix across different compute nodes/processes (process-level parallelism).

OpenMP layer (fine-grained): Responsible for accelerating locally assigned computational tasks within each MPI process by utilising multi-core resources on the node (thread-level parallelism).

3.1 How MPI Assigns Data (Coarse-Grained Parallelism)

MPI handles the **coarse-grained** parallelization and data distribution among different computing **processes** (ranks).

- **Strategy: Row-Block Decomposition** of the output matrix.
- **Implementation Details:** Within the `conv2d_stride` function, each MPI process is responsible for computing a contiguous block of rows in the final **output array** (`output`).
    - The total output height (`out_H`) is calculated.
    - The total number of rows is divided by the total number of processes (`size`) to determine the average number of rows per process (`rows_per_proc`).
    - Each process determines its local starting row (`local_start`), ending row (`local_end`), and the number of rows to compute (`local_rows`).

```
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
```

```
    int pad_top = (kH - 1) / 2;
    int pad_left = (kW - 1) / 2;

    int out_H = (H + sH - 1) / sH;
    int out_W = (W + sW - 1) / sW;

    // Distribute output rows among processes
    int rows_per_proc = (out_H + size - 1) / size;
    int local_start = rank * rows_per_proc;
    int local_end = (rank + 1) * rows_per_proc;
    if (local_end > out_H) local_end = out_H;
    int local_rows = local_end - local_start;
```

3.2 How OpenMP Parallelizes within Each Process (Fine-Grained Parallelism)

OpenMP is used for **fine-grained** parallelism **within the shared memory space** of each individual MPI process.

- **Strategy: Loop-Level Parallelism**.
- **Implementation Details:** OpenMP parallelizes the nested loops responsible for local output computation inside `conv2d_stride`.
    - The `#pragma omp parallel for` directive is applied to the two outermost loops that iterate over the local output rows (`out_i`) and all columns (`out_j`).
    - `collapse(2)` flattens the two nested loops into a single large parallel region, increasing parallelism granularity.
    - `schedule(dynamic, 16)` is used as a dynamic scheduling strategy with a chunk size of 16 output units, promoting **better load balancing** among the threads within the MPI process.

```
      // Compute local output with OpenMP parallelization
       #pragma omp parallel for schedule(dynamic, 16) collapse(2)
       for (int out_i = 0; out_i < local_rows; out_i++) {
           for (int out_j = 0; out_j < out_W; out_j++) {
               float sum = 0.0f;
               int i = (local_start + out_i) * sH;
               int j = out_j * sW;

               for (int ki = 0; ki < kH; ki++) {
                   for (int kj = 0; kj < kW; kj++) {
                       int input_i = i + ki - pad_top;
                       int input_j = j + kj - pad_left;

                       if (input_i >= 0 && input_i < H && input_j >= 0 &&
input_j < W) {
                           int local_i = input_i - input_start;
                           sum += local_f[local_i][input_j] * g[ki][kj];
                       }
                   }
               }
               output[local_start + out_i][out_j] = sum;
           }
```

```
        }
```

4. Data decomposition and distribution

Data decomposition focuses on how the **input array** (`f`) is provided to each MPI process to support its local output calculation.

- **Decomposition Target:** The **output array** (`output`) uses Row-Block Decomposition.
- **Input Data Handling (Halo/Boundary):** Due to the local nature of the convolution, each process requires an **input region larger than its assigned output region**. This necessary overlap is known as the **Halo** or boundary region.
    - The process calculates the precise starting (`input_start`) and ending (`input_end`) rows of the input array required, based on its output assignment, kernel size (`kH`), and stride (`sH`).
    - The code assumes an **SPMD (Single Program, Multiple Data)** model where the global input array (`f`) is initially accessible to all processes. Each process then allocates a local buffer (`local_f`) and uses `memcpy` to copy the required **sub-section (including the Halo)** of the input.

```
// Compute local output only if this process has rows assigned
 if (local_rows > 0) {
     // Calculate input region needed (with halo)
     int input_start = local_start * sH - pad_top;
     int input_end = (local_end - 1) * sH + kH - pad_top;

     if (input_start < 0) input_start = 0;
     if (input_end > H) input_end = H;
     int input_rows = input_end - input_start;

     // Allocate local input buffer if needed
     float **local_f = NULL;
     if (size > 1) {
         local_f = allocate_2d_array(input_rows, W);

         for (int i = 0; i < input_rows; i++) {
             memcpy(local_f[i], f[input_start + i], W * sizeof(float));
         }
     } else {
         local_f = f;
     }
```

5. Communication strategy and synchronisation

The communication strategy's primary role is to **collect** the locally computed results from all processes and ensure that all processes possess the final, complete output.

- **Mechanism: All-Broadcast Gathering**.
- **Process:**
    1. Each process independently computes its assigned output rows.
    2. All processes enter a synchronized loop, iterating through all processes p.

3. In step p, process p uses `MPI_Bcast` to broadcast its computed block of output rows to **all other processes** (including itself).
4. By the end of the loop, the series of `MPI_Bcast` calls ensures that the complete and synchronized final `output` array is available in the memory of every process.

```
// Gather results to all processes
// All processes must participate in all MPI_Bcast calls
if (size > 1) {
    for (int p = 0; p < size; p++) {
        int p_start = p * rows_per_proc;
        int p_end = (p + 1) * rows_per_proc;
        if (p_end > out_H) p_end = out_H;
        int p_rows = p_end - p_start;

        // All processes participate, even if p_rows is 0
        for (int i = 0; i < p_rows; i++) {
            MPI_Bcast(output[p_start + i], out_W, MPI_FLOAT, p, comm);
        }
    }
}
```

6. Performance Analysis

Performance was measured on **Kaya HPC** using different input sizes and thread counts.

6.1 Metrics collected:

6.1.1 Mathematical formula

Variable	Symbol	Description
Serial Execution Time	T_serial	The time taken by the single-core (1 MPI process, 1 OpenMP thread) baseline implementation of the conv2d_stride function. This is the reference point for speedup calculation.
Parallel Execution Time	T_parallel(P,T)	The time taken by the slowest MPI process to complete its computation and final synchronization/gathering. This represents the total wall-clock time for the parallel job using P MPI processes and T OpenMP threads per process.

Variable	Symbol	Formula	Description and Ideal Value
Total Number of Cores	C	$C=P\times T$	The total amount of CPU computational resources utilized by the parallel job, where P is the number of MPI processes and T is the number of OpenMP threads per process.

Variable	Symbol	Formula	Description and Ideal Value
Speedup	S(P,T)	S=T_serial/T_parallel(P,T)	Measures how many times faster the parallel version is compared to the serial baseline. Ideal Value: S≈C.
Efficiency	E(P,T)	E=S(P,T)/C×100%	Measures how effectively the total allocated resources (C) are utilized. Ideal Value: E≈100%. Efficiency less than 100% indicates overhead from communication, synchronization, or load imbalance.

6.1.2 Pure computing time analysis

In the `performance_analysis_threads` function within the `conv2d.c` file, we use the following code to specify that only pure computation time is measured:

```
      // Measure pure computation time only
        clock_gettime(CLOCK_MONOTONIC, &start);
        conv2d_omp_blocked(f, H, W, g, kH, kW, parallel_output);
        clock_gettime(CLOCK_MONOTONIC, &end);
        parallel_time = get_time_diff(start, end);
```

1. Using a high-precision timer: The code employs the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. `CLOCK_MONOTONIC` represents a monotonically increasing clock unaffected by system time changes (such as manual clock adjustments), making it ideal for performance measurement. This ensures the accuracy and consistency of timing results.
2. Tightly Enclose the Target Function: The `clock_gettime` call is placed immediately before and after the `conv2d_omp_blocked` function call. This ensures the timer records only the precise execution time from start to finish of that function.
3. Eliminate Extraneous Overhead: Through this precise timing approach, the code successfully excludes the following non-computational time overheads:

• Memory allocation: Timing occurs after the `allocate_2d_array` call, excluding memory allocation time.

• I/O operations: File read/write operations (e.g., `printf`) are excluded.

• Thread setup: The time taken by the `omp_set_num_threads` call is not included.

• Warm-up runs: The code performs an untimed “warm-up” run before the timed loop. This crucial step ensures the program code and relevant data (such as input matrices and convolution kernels) are loaded into the CPU cache. Consequently, the actual timed run avoids “cold start” effects (e.g., data loading from main memory), yielding more accurate and repeatable performance data.

6.1.3 Communication time analysis

6.1.3.1 Memory Copy Time

This portion of time is spent preparing the local input data (**Halo**) required for each MPI process.

. Logic: Each process first determines the input row range (`input_start` to `input_end`) to extract from the global input matrix f, based on its assigned output row range (`local_start` to `local_end`) and the convolution kernel size (`kH`, `kW`). This range encompasses all data required for computing the local output, including overlapping halo regions.

. Measurement Point: When `size > 1` (i.e., during multi-process execution), the code allocates a `local_f` buffer and copies the required input rows from the global `f` to the local `local_f` using `memcpy`.

```
t_comm_start = MPI_Wtime();
// ... allocate local_f
for (int i = 0; i < input_rows; i++) {
    memcpy(local_f[i], f[input_start + i], W * sizeof(float));
}
stats->memory_copy_time += MPI_Wtime() - t_comm_start;
```

Meaning: In a distributed memory environment, this memory replication represents the overhead incurred by a process preparing local data. Although it occurs within the process itself, it is fundamentally part of communication work, as it involves transforming global (or initial) data into locally computable data.

6.1.3.2 Broadcast Time

This portion of time represents the overhead incurred to synchronize and share computation results among all MPI processes.

. Logic: After all processes complete the computation of their local output rows, the code employs a series of MPI_Bcast operations to ensure each process possesses the complete final result. Each process takes turns acting as the root process, broadcasting its computed local output rows to all other processes.

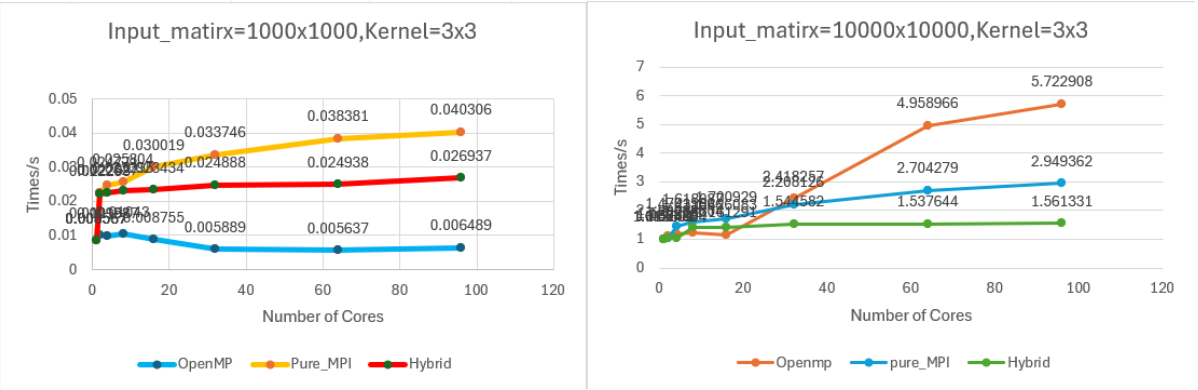
```
t_comm_start = MPI_Wtime();
for (int p = 0; p < size; p++) {
    // ... determine p_rows
    for (int i = 0; i < p_rows; i++) {
        MPI_Bcast(output[p_start + i], out_W, MPI_FLOAT, p, comm);
        stats->num_communications++;
        stats->bytes_communicated += (long long)out_W * sizeof(float);
    }
}
stats->broadcast_time = MPI_Wtime() - t_comm_start;
```

Meaning: `MPI_Bcast` is a standard MPI collective communication operation, representing the most direct communication overhead in distributed parallel computing.

6.2 Figures and charts:

6.2.1 Overall Performance Comparison and Scalability Analysis Chart

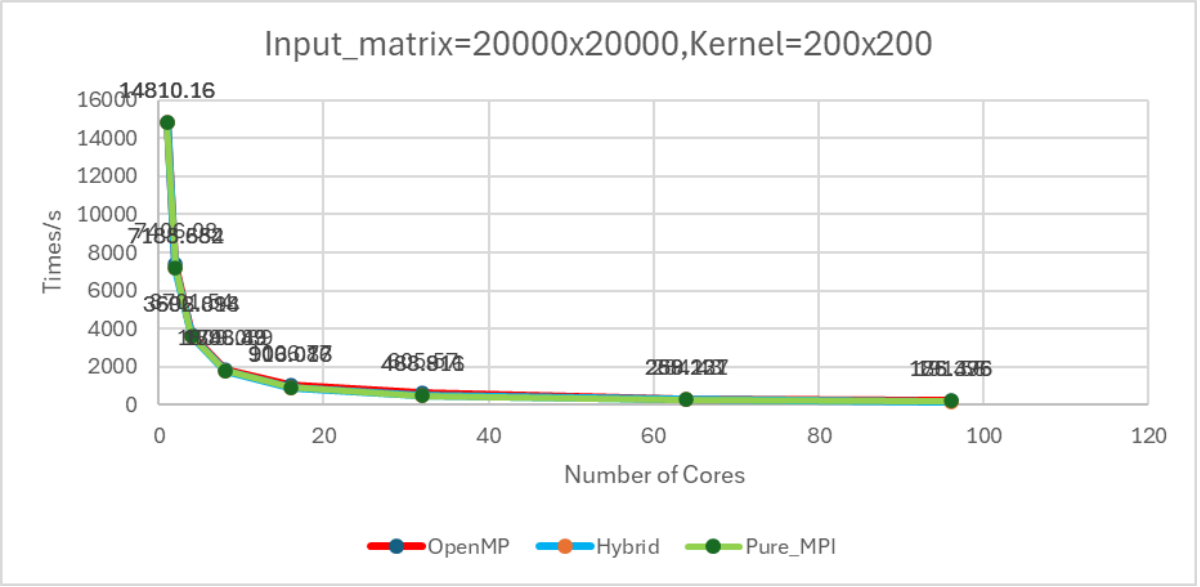
6.2.1.1 Figure 1



The two figures above illustrate the trend in total runtime as the number of cores(2,4,8,16,32,64,96) increases, using fixed input matrices of 1000x1000 and 10000x10000 with a 3x3 kernel,1x1 stride,2 nodes.

As the number of cores increases, the growth rate of communication overhead exceeds the reduction in computational time, resulting in an overall increase in total runtime. This occurs for the following reasons:

1. Communication and synchronisation overhead become dominant: MPI communication costs are prohibitively high. Increasing the number of MPI processes disproportionately amplifies inter-process communication and synchronisation overhead. Given the small problem scale, each process receives a minimal computational task (i.e., output rows). Nevertheless, each process must still execute fixed communication steps, such as copying halo data (memcpy) and collecting results (iterative `MPI_Bcast`). When computational time decreases substantially, the fixed communication latency becomes the bottleneck, and the total runtime is instead dominated by these non-computational overheads.
2. OpenMP thread management overhead: Even in pure OpenMP mode, excessive thread creation, destruction, and barrier synchronisation costs (at the end of `#pragma omp parallel for`), coupled with contention for shared caches between threads, can outweigh computational gains when thread counts are excessive.
3. Excessively Fine-Grained Tasks: The total computational load may be insufficient to offset the setup costs required for large-scale parallelisation. When tasks are divided among too many cores, the granularity per core becomes excessively fine. This results in the additional time required to initiate parallelisation exceeding the time saved through parallel computation, leading to an overall increase in runtime.

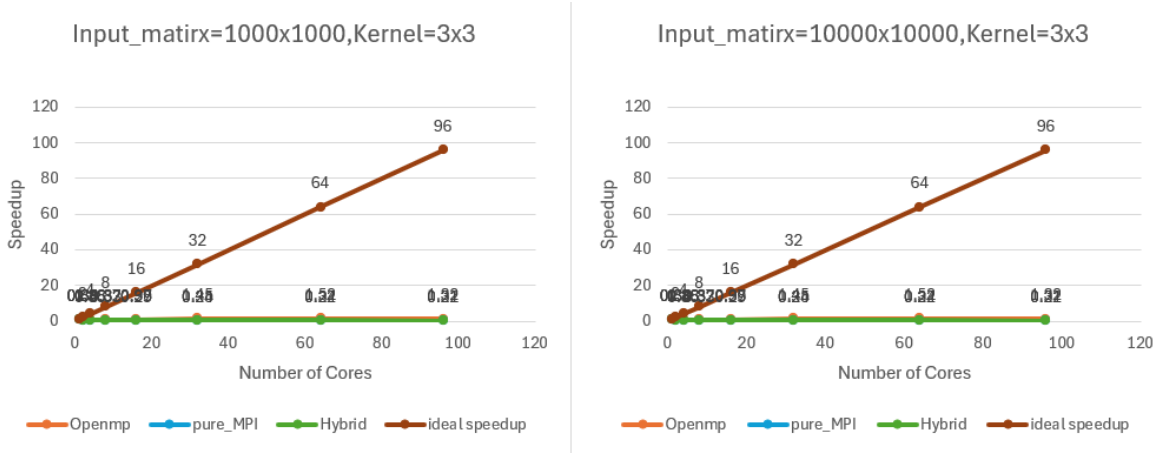


The figure above illustrate the trend in total runtime as the number of cores(2,4,8,16,32,64,96) increases, using fixed input matrices of 20000x20000 with a 200x200 kernel,1x1 stride,2 nodes.

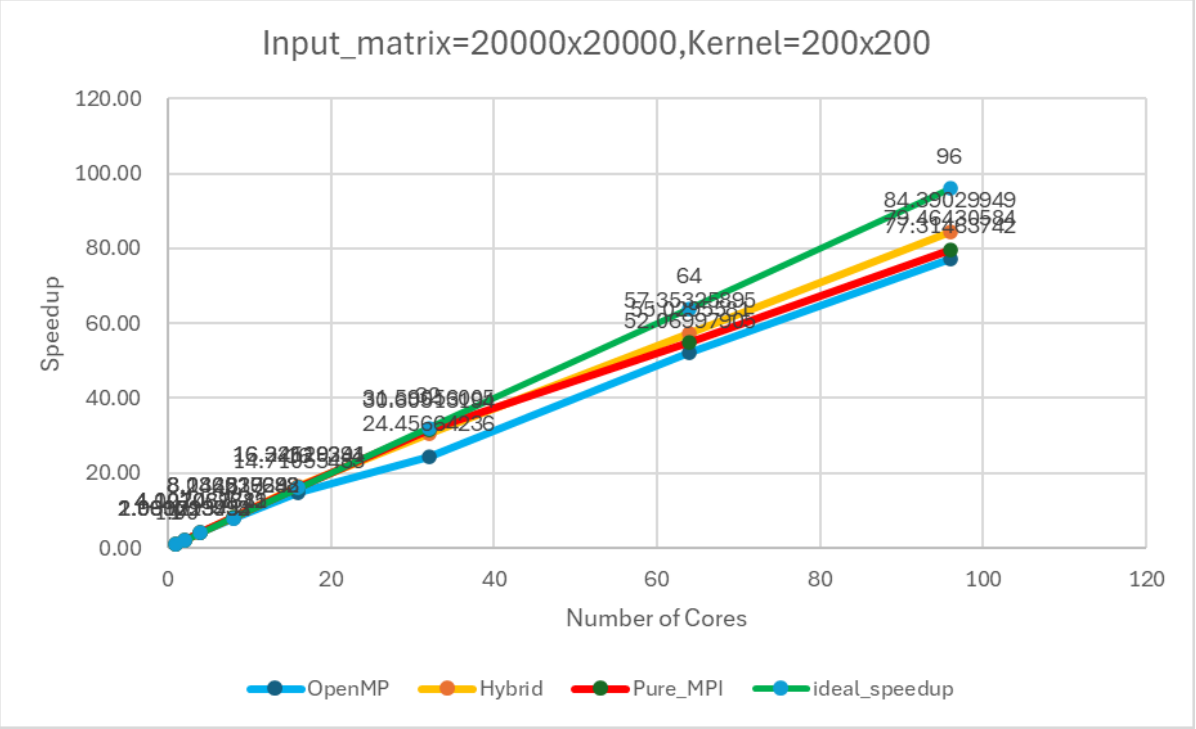
This chart powerfully demonstrates the exceptional scalability of parallel convolution implementations—including pure OpenMP, pure MPI, and MPI+OpenMP hybrid modes—when handling computationally intensive, large-scale inputs. Runtime plummets from approximately 14,810 seconds on a single core to roughly 186 seconds across 96 cores, exhibiting near-linear acceleration. Computational intensity dominates, with communication overhead effectively masked.

1. High computational intensity and acceleration effect: The 20000x20000 input matrix and 200x200 kernel matrix generate an enormous computational workload (FLOPs). The substantial computational gains fully offset the communication, synchronisation, and management overhead introduced by parallelisation, thereby achieving the efficient acceleration sought in high-performance computing.
2. Low Communication/Computational Ratio: Given the substantial computational task granularity assigned to each core, the growth in communication volume (Halo Exchange) and synchronisation frequency (MPI_Bcast loops) required for boundary data exchange and result collection remains minimal relative to the total computational load.
3. Pattern Performance Convergence: The OpenMP, Hybrid, and Pure_MPI curves remain remarkably close across the entire test range (from 1 to 100 cores). This indicates that in this compute-bound scenario, computational time constitutes the primary bottleneck, rendering communication overhead—whether synchronisation in shared memory or MPI messaging in distributed memory—negligible. This validates that the row-blocking and loop-parallelisation strategy adopted for tackling large-scale problems is both efficient and correct.

6.2.1.2 Figure 2



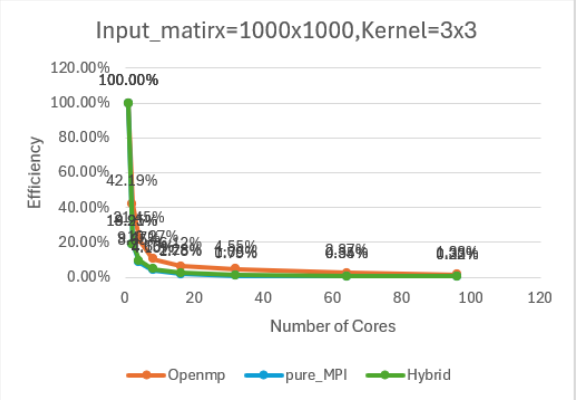
The figure above illustrates that parallel overhead dominates, with insufficient computational gains leading to failed acceleration. Whether for 1000x1000 or 10000x10000 inputs (right-hand figure, where despite a 100-fold increase in scale, the 3x3 core configuration still yields low overall computation), the total computational intensity remains insufficient to offset the fixed overhead introduced by parallelisation. The acceleration ratios for OpenMP, Pure_MPI, and Hybrid remain at extremely low levels across nearly the entire curve, levelling off rapidly. This indicates that for problems of relatively low computational intensity, increasing the number of cores only disproportionately amplifies the overhead of communication, synchronisation, and thread management. Communication latency and startup costs become the primary bottlenecks, preventing effective acceleration ratios from being realised.

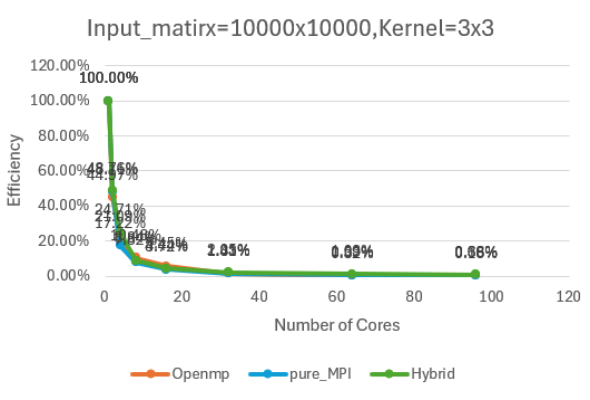


The input of 20000 x 20000 combined with the large kernel of 200 x 200 generated an exceptionally massive computational workload (with computational intensity increasing tens of thousands of times compared to Figure 1). The substantial computational gains were sufficient to fully offset all communication and synchronisation overhead introduced by parallelisation. Consequently, all three parallelisation approaches achieved significant and near-linear speedup.

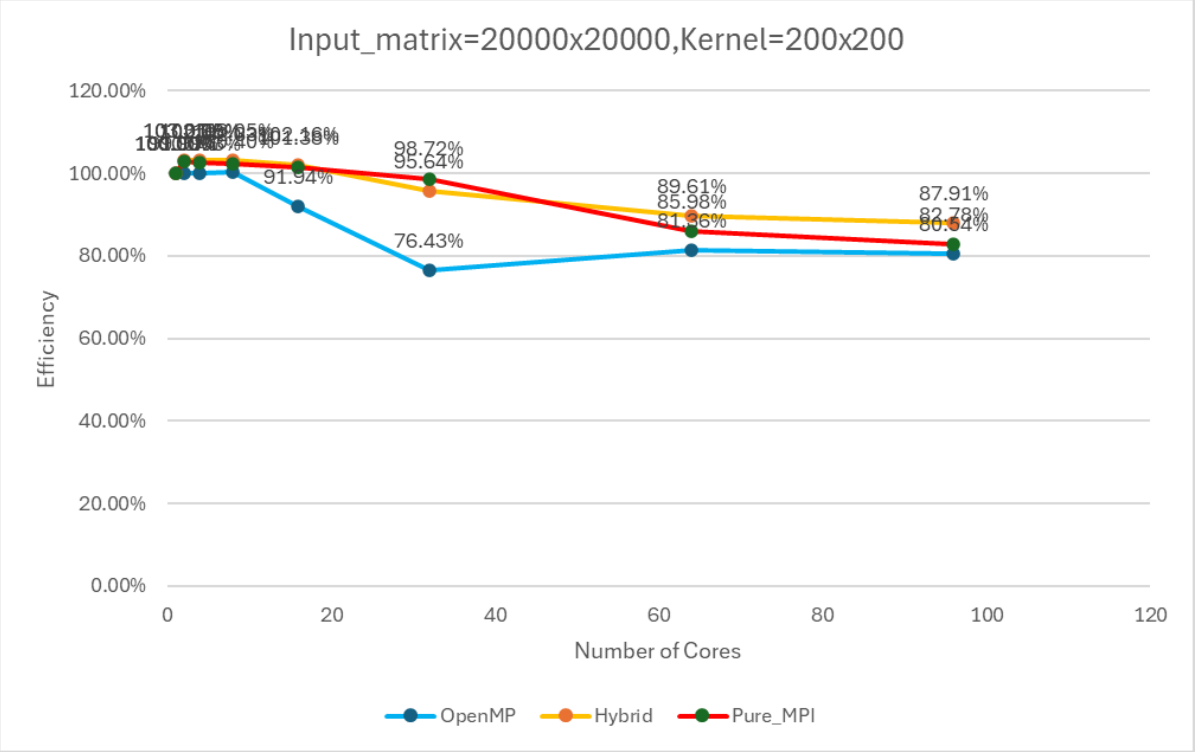
In computationally intensive scenarios such as20000 x 20000, the performance of pure MPI mode is marginally lower due to its inherent high communication latency (though computationally intensive, latency still persists). The Hybrid mode successfully balances computational and communication overhead by employing OpenMP for efficient intra-node computation while minimising inter-node communication through fewer MPI processes. This equilibrium enables the Hybrid mode to utilise all cores more effectively, thereby achieving performance closer to the ideal speedup ratio.

6.2.1.3 Figure 3





These two figures illustrate the efficiency changes for the aforementioned 1000x1000 and 10000x10000 configurations. They demonstrate that parallelization overhead (communication, synchronization, thread management) consumes the vast majority of runtime. Since each core receives an extremely small computational workload, the fixed overhead of parallel initialization and the latency of each communication synchronization cannot be effectively offset by the computational work. Consequently, adding each additional core yields computational gains far outweighed by the introduced system overhead, resulting in extremely low resource utilization. This algorithm is unsuitable for large-scale parallelization at this scale.



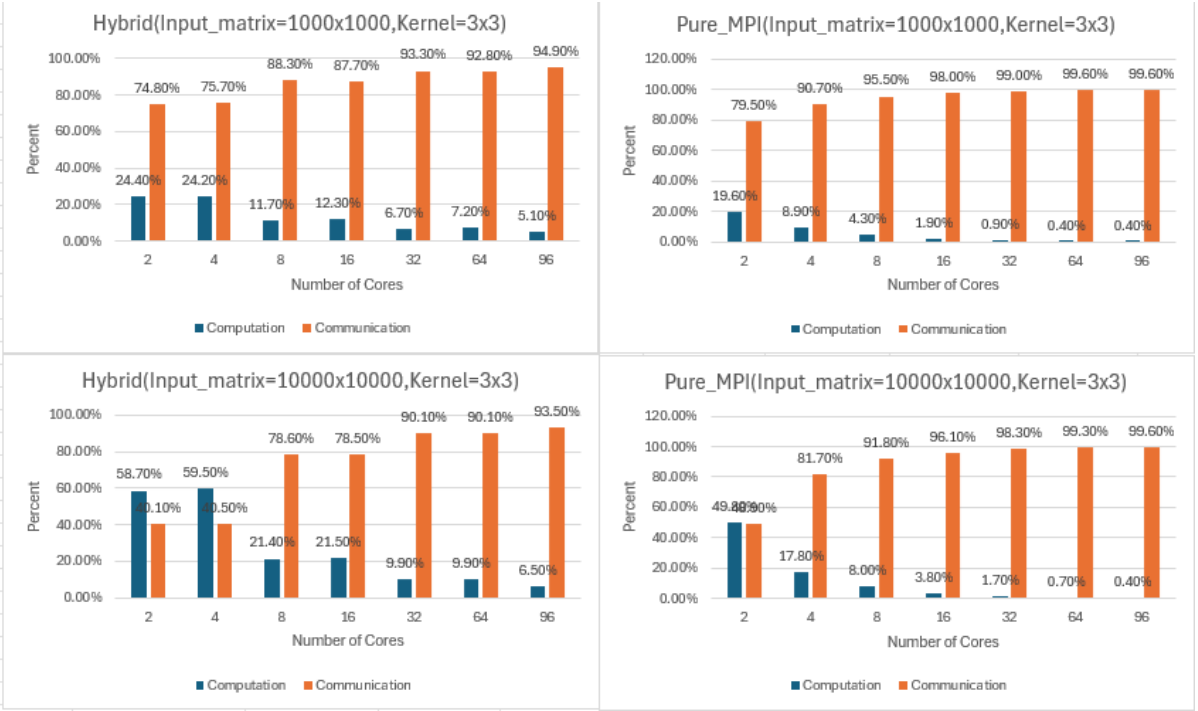
This figure represents the efficiency of a 20000x20000 input matrix and a 200x200 kernel. It shows that despite the increase in core count, efficiency (E) remains at a high level (e.g., 70%–85%) with a relatively gradual decline. This indicates that the massive computational workload effectively masks the parallel overhead. The high computational intensity ensures that computation time

remains the dominant factor, with system overhead accounting for a negligible proportion. The algorithm maintains high granularity and excellent load balancing even as the number of cores increases. Consequently, it demonstrates outstanding parallel scalability, making it highly suitable for operation in large-scale HPC environments featuring multi-core processors and multiple nodes.

Hybrid achieves peak efficiency by striking the optimal balance between communication overhead and thread contention. It leverages MPI's cross-node capabilities to attain higher speedup than OpenMP, while utilizing OpenMP's low-latency shared memory to achieve greater efficiency than Pure_MPI. This represents the optimal strategy for running massively parallel programs on HPC clusters.

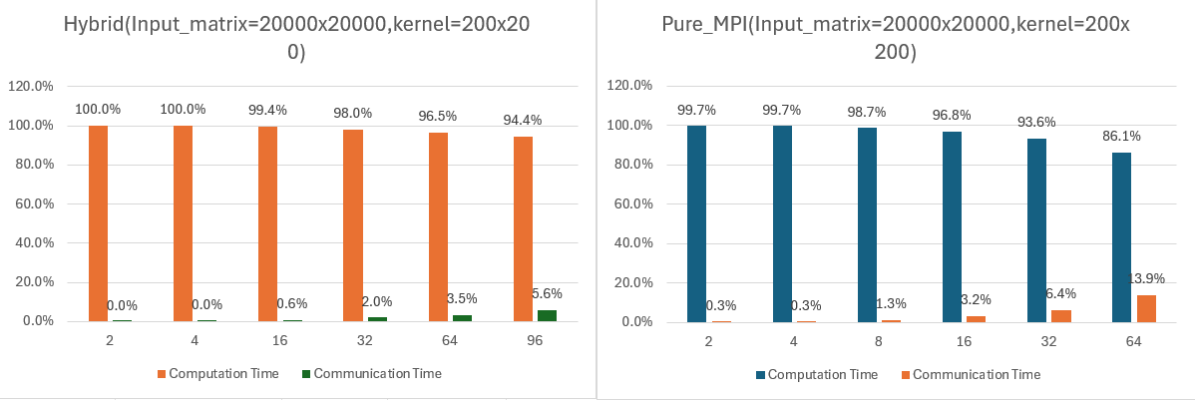
6.2.2 Communication/Computational Overhead Analysis Chart

6.2.2.1 Figure 1



The figure above shows the proportion of computation time versus communication time for hybrid and pure MPI implementations across different input matrices (1000x1000, 10000x10000) and 3x3 cores at varying core counts. It is evident that communication time significantly outweighs computation time in both scenarios. This explains why the total runtime, acceleration ratio, and efficiency variations did not achieve optimal results under these conditions.

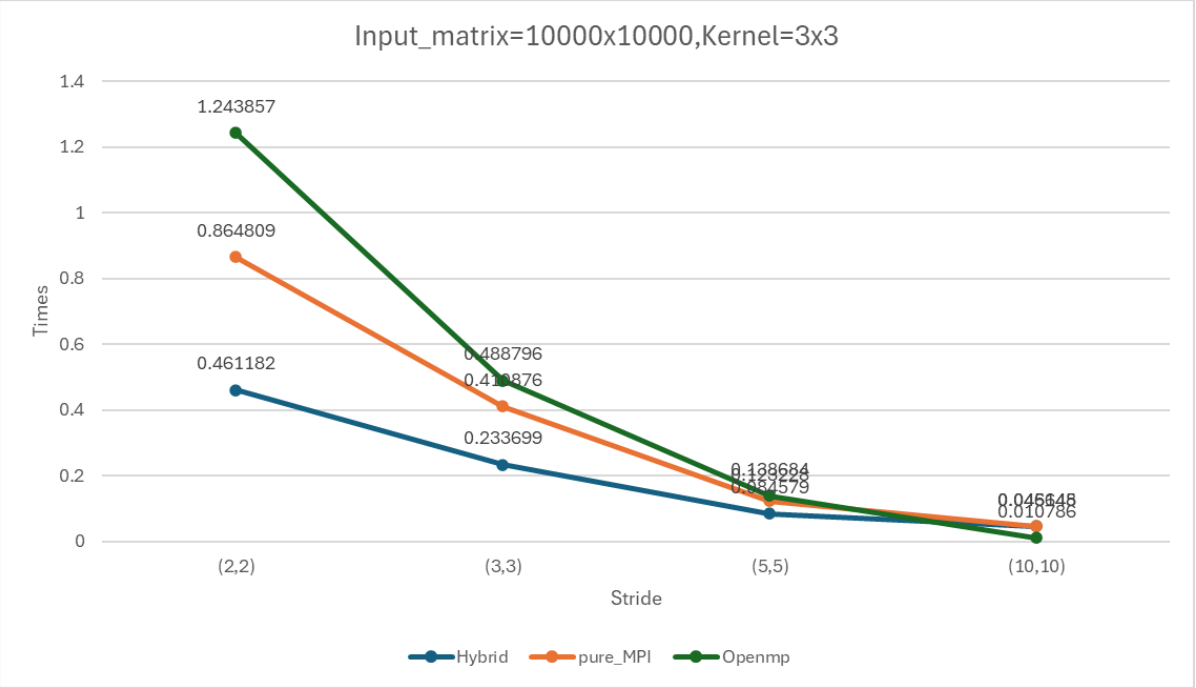
6.2.2.2 Figure 2



The figure above shows the proportion of computation time versus communication time for hybrid and pure MPI implementations on different input matrices (20000x20000) and 3x3 cores across varying core counts. It is evident that in both scenarios, computation time significantly outweighs communication time. This indicates that for large-scale matrices under these conditions, the total runtime, acceleration ratio, and efficiency variations have reached an ideal state.

6.2.3 Stride Influence Analysis Chart

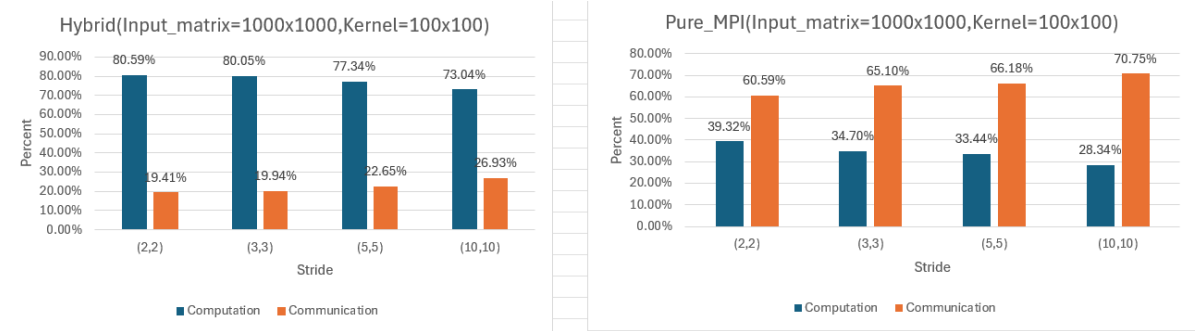
6.2.3.1 Figure 1



The figure above shows the variation in total runtime for hybrid, pure MPI, and OpenMP implementations using the same input matrix (10000x10000) and a 3x3 core configuration across 2 nodes at different stride lengths. It demonstrates that as the stride increases, the total runtime decreases significantly.

This demonstrates that Hybrid mode performs best in computationally intensive scenarios (small strides) due to its load balancing advantages. For instance, at stride (2,2), it took 1.244 seconds—significantly outperforming OpenMP's 1.387 seconds. However, as computational intensity decreases with increasing stride, OpenMP gradually gains an advantage by avoiding cross-node communication overhead. At stride (10,10), it completes in just 0.047 seconds—approximately 66% faster than Hybrid's 0.139 seconds. Pure MPI consistently lags due to communication overhead. This trend demonstrates that the optimal parallelization strategy must be dynamically selected based on specific computational intensity.

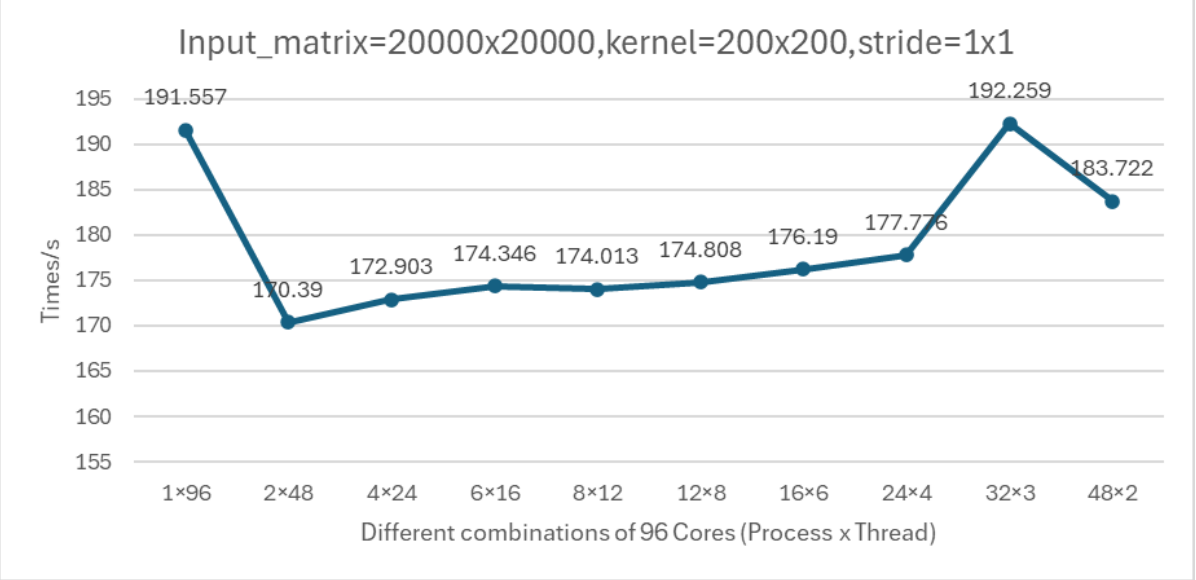
6.2.3.2 Figure 2



This image analysis compares the Hybrid and the Pure MPI under different Stride settings(same input matrix and kernel,2 nodes). Analysis reveals that the Hybrid model significantly outperforms the Pure MPI model. For the Hybrid model, computation time consistently dominates (73%–81%), with communication overhead being relatively minor (19%–27%). The communication time proportion increases slightly as stride lengthens. In contrast, the Pure MPI model treats communication as the primary bottleneck, with communication time accounting for as much as(61%-71%) of the total time, while effective computation time only constitutes (28%-39%). More critically, as the stride increases from (2, 2) to (10, 10), the communication overhead share in the Pure MPI model increases significantly while computational efficiency declines markedly. This strongly demonstrates that the Hybrid model can more effectively utilize computational resources for such convolution tasks by reducing expensive cross-node communication.

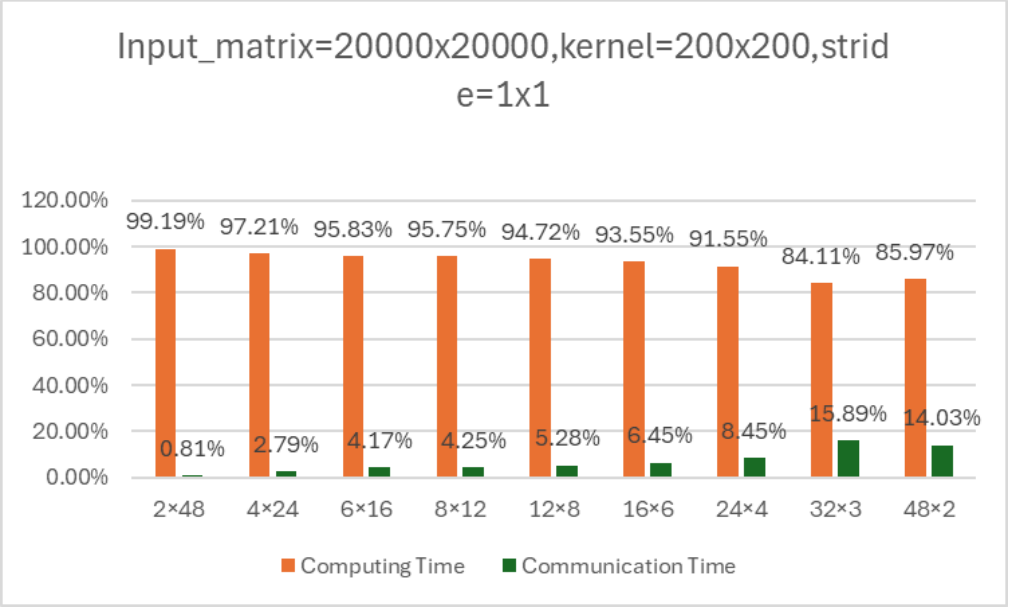
6.2.4 Optimal Configuration Analysis Chart for Hybrid Parallel Systems(2 nodes)

6.2.4.1 Figure 1



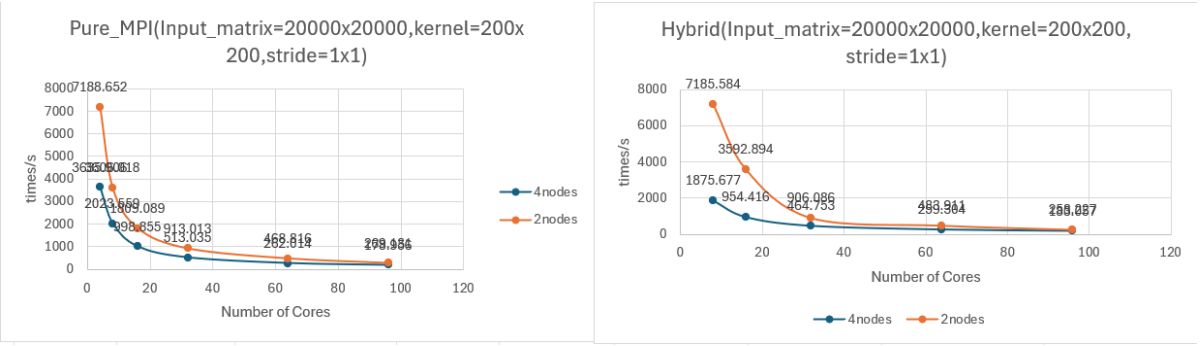
This chart clearly illustrates the performance of various combinations of MPI processes and OpenMP threads across 96 compute cores. Results show that the 2×48 configuration achieved the shortest total runtime of 170.39 seconds, ranking first with an 11.1% improvement over the pure OpenMP approach (191.56 seconds). The optimal performance range is concentrated between the 2×48 and 12×8 configurations, with a total time difference of only 2.6%. This indicates that a moderate number of MPI processes (2–12) can effectively leverage the advantages of hybrid parallelism. However, when the number of MPI processes increases further to 32×3 and 48×2, performance deteriorates significantly to 183.72 seconds, demonstrating that excessive processes introduce severe performance overhead.

6.2.4.2 Figure 2



This figure provides an in-depth analysis of the ratio between computation and communication time across different configurations, clearly elucidating the underlying causes of performance variations. The 2×48 configuration achieves optimal balance with 99.2% computation and only 0.8% communication overhead. As the number of MPI processes increases, communication overhead surges sharply from 0.8% to 15.9% in the 32×3 configuration. Within the optimized range from 4×24 to 12×8, communication overhead remains below 5.3% while computational efficiency stays above 94.7%. However, beyond 16 MPI processes, the rapid increase in communication overhead completely offsets the gains in computational efficiency, leading to a decline in overall performance.

6.2.5 Performance charts for Pure_MPI and Hybrid mode across different nodes



This chart compares the scalability performance of Pure MPI and Hybrid mode across 2-node and 4-node environments. Overall, both parallel modes demonstrate strong scalability, with performance increasing nearly linearly as the number of cores grows. Hybrid mode slightly outperforms Pure MPI in most configurations, particularly within the medium core range (32-64 cores). At the maximum scale (96 cores), their performance converges, indicating that the hybrid parallel mode more effectively balances computational and communication overhead, yielding sustained performance advantages in multi-node scaling.

At identical core counts, the 4-node configuration consistently demonstrates superior performance compared to the 2-node setup. Specifically, at 96 cores, the 4-node Hybrid mode completed in 183.06 seconds, 0.4% faster than the 2-node's 183.72 seconds. At 64 cores, the difference was more pronounced: the 4-node Pure MPI achieved 262.01 seconds, outperforming the 2-node's 269.13 seconds. This advantage primarily stems from the 4-node architecture's

superior ability to distribute communication load and reduce resource contention within individual nodes. Performance gains are most pronounced in the mid-range core count (32-64 cores), where the 4-node Hybrid configuration outperformed the 2-node setup by 4.1% at 32 cores. However, as core counts increase further, cross-node communication overhead begins to dominate performance, causing the two node configurations to converge in performance at the highest core counts.

7. Conclusion

This experiment systematically validated the significant advantages of the MPI+OpenMP hybrid parallelism model in large-scale two-dimensional convolution computations through comprehensive performance testing and analysis. Experimental results demonstrate that in computationally intensive scenarios (e.g., 20000×20000 matrix with 200×200 convolution kernel), the 2×48 hybrid configuration achieved optimal performance with a minimum runtime of 170.39 seconds—representing an 11.1% improvement over pure OpenMP. This advantage stems from effectively balancing computational load and communication overhead through an optimal number of MPI processes (2–12). As problem scale varies, the optimal parallelization strategy requires dynamic adjustment—hybrid parallelism suits computationally intensive tasks, while pure OpenMP is more suitable for computationally light tasks. Additionally, the 4-node architecture demonstrates superior scalability within the medium core range through better communication load distribution.