Distribuited Stencil Computation

A Case Study with OpenMP and MPI

Andrea Spinelli

25/07/2025

University of Trieste

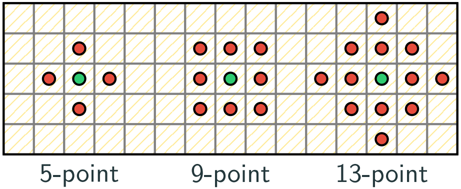
Introduction

Problem Statement: Stencil Computation

The objective of this project is to implement a parallel **5-point stencil** computation utilizing OpenMP and MPI.

Stencil computations have diverse applications including the numerical solution of partial differential equations, such as the ***heat equation***.

- Each grid point $x_{ij}(t)$ is updated iteratively based on its own value at the previous time step, $x_{ij}(t-1)$, and the values of its ***neighbors***.

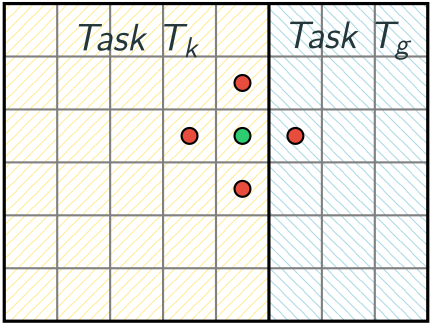


In this work, we focus on the 5-point stencil.

For large grids, the computational cost becomes significant, motivating the need for ***parallelization***.

Challenges: Data Dependencies & Communication

To enable parallel execution, the computational grid can be partitioned into subdomains, with each subdomain assigned to a separate process.



However, a key challenge arises: each grid point depends on the values of its neighbors, including those located in adjacent subdomains.

This introduces data dependencies across process boundaries and necessitates inter-process communication.

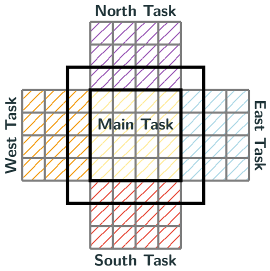
Challenges: Efficient Communication

Since processes must exchange border data at each iteration, we need the processes to communicate with each other.

The idea is to enlarge the local patches to include the ***halo regions*** (the border of the local patch).

Communication can be expenive if the problem is not well implemented:

- efficient communication pattern
  $\rightarrow$ Non-Blocking Communication
- efficient memory management
  $\rightarrow$ Avoid unnecessary allocations and copies
  $\rightarrow$ Cache locality among processes
- Computation / Communication Overlap
  (we will see further)



Implementation

Parallelization Approach: Hybrid MPI + OpenMP

The implementation tackles both inter-node and intra-node parallelism:

Distributed-memory parallelism (MPI):

- The grid is decomposed into patches, each assigned to a process.
- At each iteration, processes exchange halo data with their neighbors using non-blocking communication (`MPI_Isend`, `MPI_Irecv`).
- Collective operations (`MPI_Reduce`, `MPI_Bcast`, `MPI_Gather`) are used for global statistics and data distribution.

Shared-memory parallelism (OpenMP):

- Within each MPI process, OpenMP parallelizes the main update loops (`#pragma omp parallel for`).
- Reductions (e.g., for total energy) are performed in parallel.

Optimization: Overlapping Communication and Computation

In stencil codes, processes must wait for halo data from neighbors before updating boundary points. We can potentially improve performance by **overlapping communication with computation**.

We split the update loop into two parts, ***interior*** and ***border***:

Optimization: Overlapping Communication and Computation

In stencil codes, processes must wait for halo data from neighbors before updating boundary points. We can potentially improve performance by **overlapping communication with computation**.

We split the update loop into two parts, ***interior*** and ***border***:

1. **Initiate Non-Blocking Communication**: Start sending and receiving halo data using `MPI_Isend` and `MPI_Irecv`.

Optimization: Overlapping Communication and Computation

In stencil codes, processes must wait for halo data from neighbors before updating boundary points. We can potentially improve performance by **overlapping communication with computation**.

We split the update loop into two parts, ***interior*** and ***border***:

1. **Initiate Non-Blocking Communication**: Start sending and receiving halo data using `MPI_Isend` and `MPI_Irecv`.
2. **Compute the Interior**: While the halo data is in transit over the network, compute the new values for the ***interior points*** of the local grid. These points do not depend on the halo data.

Optimization: Overlapping Communication and Computation

In stencil codes, processes must wait for halo data from neighbors before updating boundary points. We can potentially improve performance by **overlapping communication with computation**.

We split the update loop into two parts, ***interior*** and ***border***:

1. **Initiate Non-Blocking Communication**: Start sending and receiving halo data using `MPI_Isend` and `MPI_Irecv`.
2. **Compute the Interior**: While the halo data is in transit over the network, compute the new values for the ***interior points*** of the local grid. These points do not depend on the halo data.
3. **Synchronize and Complete**: Use `MPI_Waitall` to ensure all halo communications have completed.

Optimization: Overlapping Communication and Computation

In stencil codes, processes must wait for halo data from neighbors before updating boundary points. We can potentially improve performance by **overlapping communication with computation**.

We split the update loop into two parts, ***interior*** and ***border***:

1. **Initiate Non-Blocking Communication**: Start sending and receiving halo data using `MPI_Isend` and `MPI_Irecv`.
2. **Compute the Interior**: While the halo data is in transit over the network, compute the new values for the ***interior points*** of the local grid. These points do not depend on the halo data.
3. **Synchronize and Complete**: Use `MPI_Waitall` to ensure all halo communications have completed.
4. **Compute the Border**: Once the halo data is available, compute the new values for the ***border points*** of the local grid.

Overlapping: Consequences

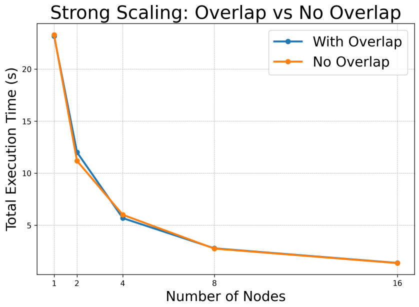
Contrary to the hypothesis, this optimization yielded no significant improvements because the workload is heavily **compute-bound**.

The time spent in calculations vastly outweighs communication time.

For example, in an 8-node run, the breakdown was:

Computation Time:	**$\sim$96%**
Communication Time:	**$\sim$4%**

Hiding the negligible communication latency provides no visible benefit.



Overlapping is effective only when communication latency is a significant performance bottleneck.

Implementation Details: Memory and Data Management

Other techniques were used to ensure performance and correctness.

Efficient Buffer Management

To minimize memory allocation overhead and avoid unnecessary copies:

- Communication buffers are ***pre-allocated*** once at initialization and reused throughout the simulation.
- For contiguous boundaries (North/South), direct pointer arithmetic is used to send/receive data ***in-place*** without intermediate buffering.
- Only non-contiguous boundaries require temporary buffers.

NUMA-Aware Memory Allocation

The ***first-touch policy*** on NUMA architectures is exploited. By parallelizing the initialization of the grid with OpenMP, we ensure that each thread allocates the memory pages for its assigned rows on the NUMA node it is running on. This minimizes remote memory access during the computation phase.

Build Process and Compilation

Compilation is handled by a `Makefile` that leverages the `mpicc` wrapper.

Compiler Flags

Key flags used for the final performance runs:

- `-Ofast`: Aggressive optimization, enables `-O3` optimizations and more
- `-flto`: Link-time optimization for improved performance.
- `-fopenmp`: Enables OpenMP support for parallel regions.
- `-march=native`: Generates code optimized for the build machine's architecture.
- `-Wall -Wextra -Wpedantic -Wshadow -Wuninitialized`: Enables a wide range of compiler warnings for code quality.

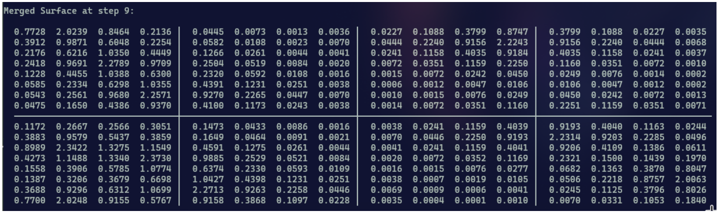
Note on Compilation Environment

Compile on a compute node to ensure `-march=native` targets the correct architecture (login node may differ).

Verbose Output and Instrumentation

The code uses preprocessor directives to control the level of runtime output via compile-time flags:

- `VERBOSE_LEVEL=1` (`verbose1` target): Prints neighbor information, timing statistics, and energy statistics at each step.
- `VERBOSE_LEVEL=2` (`verbose2` target): In addition to level 1, prints the full grid state at each iteration for detailed inspection.



```
Merged Surface at step 9:

  0.7728  2.0239  0.8464  0.2136 |  0.0445  0.0073  0.0013  0.0036 |  0.0227  0.1088  0.3799  0.8747 |  0.3799  0.1088  0.0227  0.0035
  0.3912  0.9871  0.6048  0.2254 |  0.0582  0.0108  0.0023  0.0070 |  0.0444  0.2240  0.9156  2.2243 |  0.9156  0.2240  0.0444  0.0068
  0.2176  0.6216  1.0350  0.4449 |  0.1266  0.0261  0.0044  0.0041 |  0.0241  0.1158  0.4035  0.9184 |  0.4035  0.1158  0.0241  0.0037
  0.2418  0.9691  2.2789  0.9709 |  0.2504  0.0519  0.0084  0.0020 |  0.0072  0.0351  0.1159  0.2250 |  0.1160  0.0351  0.0072  0.0010
  0.1228  0.4455  1.0388  0.6300 |  0.2320  0.0592  0.0108  0.0016 |  0.0015  0.0072  0.0242  0.0450 |  0.0249  0.0076  0.0014  0.0002
  0.0585  0.2334  0.6298  1.0355 |  0.4391  0.1231  0.0251  0.0038 |  0.0006  0.0012  0.0047  0.0106 |  0.0106  0.0047  0.0012  0.0002
  0.0543  0.2561  0.9680  2.2571 |  0.9270  0.2265  0.0447  0.0070 |  0.0010  0.0015  0.0076  0.0249 |  0.0450  0.0242  0.0072  0.0013
  0.0475  0.1650  0.4386  0.9370 |  0.4100  0.1173  0.0243  0.0038 |  0.0014  0.0072  0.0351  0.1160 |  0.2251  0.1159  0.0351  0.0071
  -------------------------------+---------------------------------+---------------------------------+--------------------------------
  0.1172  0.2667  0.2566  0.3051 |  0.1473  0.0433  0.0086  0.0016 |  0.0038  0.0241  0.1159  0.4039 |  0.9193  0.4040  0.1163  0.0244
  0.3883  0.9579  0.5437  0.3859 |  0.1649  0.0464  0.0091  0.0021 |  0.0070  0.0446  0.2250  0.9193 |  2.2314  0.9203  0.2285  0.0496
  0.8989  2.3422  1.3275  1.1549 |  0.4591  0.1275  0.0261  0.0044 |  0.0041  0.0241  0.1159  0.4041 |  0.9206  0.4109  0.1386  0.0611
  0.4273  1.1488  1.3340  2.3730 |  0.9885  0.2529  0.0521  0.0084 |  0.0020  0.0072  0.0352  0.1169 |  0.2321  0.1500  0.1439  0.1970
  0.1558  0.3906  0.5785  1.0774 |  0.6374  0.2330  0.0593  0.0109 |  0.0016  0.0015  0.0076  0.0277 |  0.0682  0.1363  0.3870  0.8047
  0.1387  0.3206  0.3679  0.6698 |  1.0427  0.4398  0.1231  0.0251 |  0.0038  0.0007  0.0019  0.0105 |  0.0506  0.2218  0.8757  2.0063
  0.3688  0.9296  0.6312  1.0699 |  2.2713  0.9263  0.2258  0.0446 |  0.0069  0.0009  0.0006  0.0041 |  0.0245  0.1125  0.3796  0.8026
  0.7700  2.0248  0.9155  0.5767 |  0.9158  0.3868  0.1097  0.0228 |  0.0035  0.0004  0.0001  0.0010 |  0.0070  0.0331  0.1053  0.1840
```

This approach ensures that verbose output and measurement code are excluded from the final performance binaries unless explicitly requested, avoiding unnecessary overhead.

Reproducibility

A key goal is to ensure the parallel code is **reproducible and bitwise compatible with the serial version**, allowing reliable verification.

This is challenging, especially for source generation. In a naive parallel approach, each process generates sources independently, which cannot match the serial version.

To ensure compatibility:

- All sources are generated by the rank 0 process and then distributed to the other processes.
- A `-s` (seed) flag sets the random seed, so the same input always produces the same results.

Results

Overview of Scaling Tests

To comprehensively evaluate performance, three principal types of scalability tests were conducted:

- **OpenMP thread scaling**:
  single MPI task,
  varying threads (1, 2, 4, 8, 16, 32, 56, 84, 112)
- **MPI strong scaling**:
  fixed problem size,
  increasing nodes (1, 2, 4, 8, 16)
- **MPI weak scaling**:
  fixed workload per node,
  increasing nodes (1, 2, 4, 8, 16) and grid size accordingly

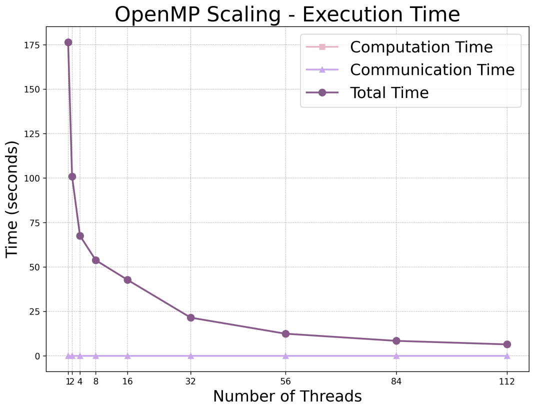
Batch scripts were used to automate runs and collect timing data.

For each configuration, ***total***, ***computation***, and ***communication*** times were measured.

OpenMP Scaling: Execution Time

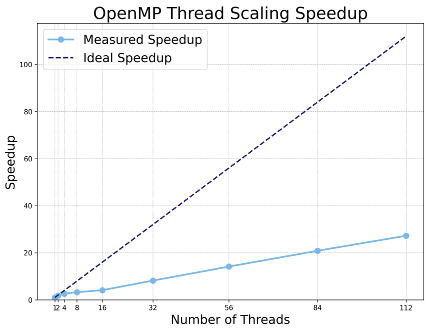
OpenMP thread scaling: single MPI task, varying number of threads.

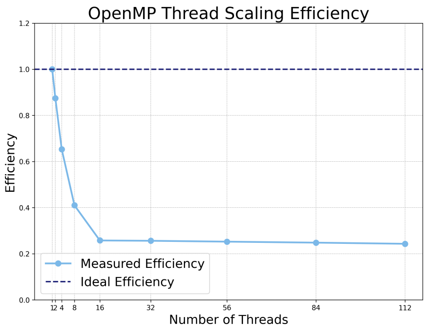
Measures how execution time changes as more threads are used



Execution time decreases with more threads, but the benefit diminishes at high thread counts due to memory bandwidth and thread overhead.

OpenMP Scaling: Speedup and Efficiency



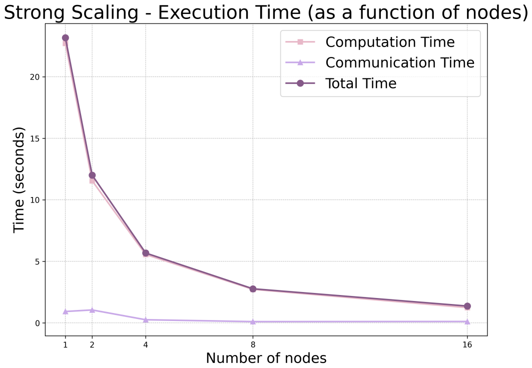


- Speedup is sublinear for large thread counts; best performance is achieved with moderate thread numbers.
- Efficiency drops as the number of threads increases, mainly due to memory bandwidth limitations and thread management overhead.

MPI Strong Scaling: Execution Time

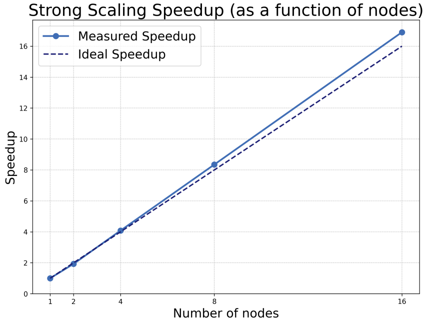
Strong scaling: fixed problem size, increasing number of nodes.

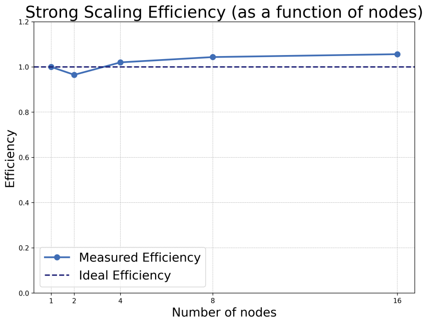
Measures how execution time decreases as more resources are used for the same total work.



Execution time drops as more nodes are used.

MPI Strong Scaling: Speedup and Efficiency





Ideally, speedup should be linear.

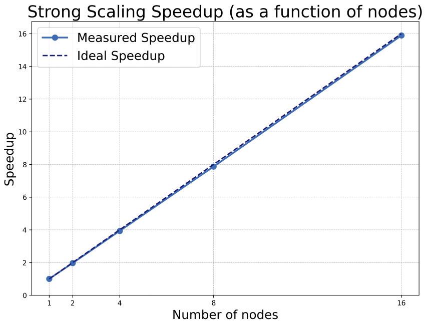
Results show a **superlinear speedup**, with efficiency greater than 1.

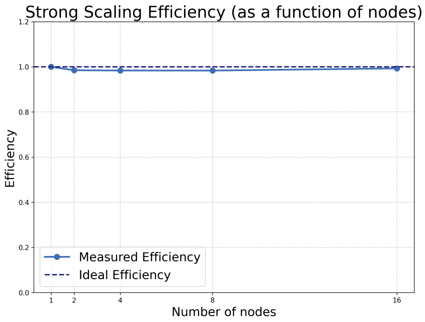
This means the parallel code is more than N times faster on N nodes, which seems to violate Amdahl's Law. This is not an error, but (probably) a real phenomenon caused by **cache effects**.

MPI Strong Scaling: Superlinear Speedup Explained

Superlinear speedup can occur due to improved cache utilization as the local problem size per node decreases.

To verify this, strong scaling was repeated with only one MPI task per node, removing the cache benefit from multiple tasks sharing a node.



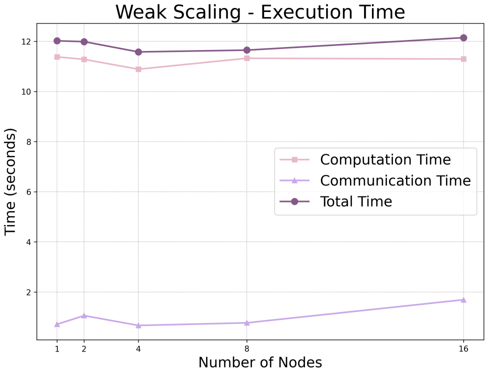


In this configuration, the superlinear effect disappears and efficiency returns to below 1, confirming the cache explanation.

MPI Weak Scaling: Execution Time

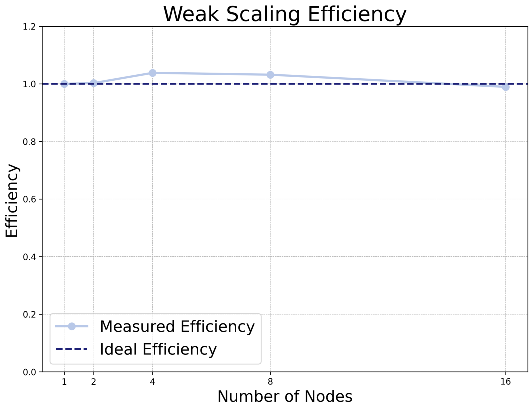
Weak scaling: fixed workload per node, increasing number of nodes.

Measures if execution time remains constant as the problem size grows proportionally with resources.



Ideally, execution time should remain constant. Observed times are nearly flat, with minor variations due to system effects.

MPI Weak Scaling: Efficiency



- Efficiency stays near 1, as expected for good weak scaling.
- Observed efficiency slightly above 1 in some cases, likely due to cache effects or measurement noise.

Conclusions

Key Findings and Conclusions

The parallel stencil solver, using a hybrid MPI+OpenMP approach, demonstrated strong scalability and high efficiency on a modern HPC.

Main Achievements

- **Strong and Weak Scaling:**
  The code achieved (over) ideal strong and weak scaling, confirming its effectiveness for large-scale workloads.
- **Hybrid Parallelism:**
  Combining OpenMP (for shared-memory parallelism within nodes) and MPI (for distributed-memory parallelism across nodes) provided robust performance and flexibility.

Key Takeaway

For compute-bound applications on modern HPC systems, maximizing data locality and cache usage can deliver performance gains that exceed predictions from simple parallel scaling laws.

Future Work and Potential Enhancements

Building on this solid foundation, several avenues for further optimization and exploration can be pursued:

Algorithmic & Communication

- **Implement MPI Derived Datatypes**: Replace manual packing/unpacking of non-contiguous halo data (East/West boundaries) with `MPI_Type_vector`.
- **Advanced Domain Decomposition**: Implement dynamic load balancing strategies, such as recursive bisection, to handle non-uniform workloads (e.g., simulations with localized sources).

Architecture & Performance Tuning

- **GPU Acceleration**: Port the `update_plane` kernel to a GPU using OpenMP or CUDA to exploit massive data parallelism.
- **Deeper NUMA Analysis**: Experiment with different thread/task placement strategies using SLURM's binding options.

Thank You!