



Utkarsh Pathak, Maxim Barnstorf

2D Parallel Seismic Wave Forward Modeling

Marmousi Model (MPI + OpenMP)

Table of contents

- 1 Motivation
- 2 Sequential Solution
- 3 Parallelization Strategy
- 4 Performance Gain
- 5 Conclusion

Motivation

■ Why seismic wave forward modeling?

- ▶ Requires high computational resources.
- ▶ Spatially decomposable.
 - Perfect for domain decomposition (MPI) and shared-memory threading (OpenMP)
- ▶ Readily available real-world datasets. [2, 2025]
- ▶ Good documentation of the algorithm exists.

■ Goal:

- ▶ Understand and demonstrate how to parallelize scientific code across nodes (MPI) and within nodes (OpenMP), how it scales on an actual cluster and how to effectively benchmark it.

Problem Description

- 2D **elastic** wave equation using stress and velocity.
- Marmousi2 model: Elastic Marmousi2 model with separate P-velocity, S-velocity and density, each with size $13601 * 2801$ and a total of 38.1 million points.
- From this, we derive fields particle velocities v_x, v_z ; stress components $\sigma_{xx}, \sigma_{zz}, \sigma_{xz}$. [1, 2006]
- The velocity and stress components at each point will be calculated for each timestep, based on the previous values. Ultimately, we are mainly interested in the v_z , which we use to visualize the simulation.

Algorithm

Core time-stepping loop

```
for it in range(n_iterations):  
    inject_ricker_source(sxx, szz, it)      # stress source  
    update_stress_c(vx, vz, sxx, szz, sxz, # C kernel  
                    lam, lam2mu, mu, damp, dt, dx, dz)  
    update_velocity_c(vx, vz, sxx, szz, sxz, # C kernel  
                     inv_rho, damp, dt, dx, dz)  
    if it % frame_stride == 0:  
        write_frame_to_hdf5(vz)
```

While the program is mainly written in Python, the C (`libelastic_kernels_seq.so`) is used to improve performance and to allow for OpenMP to be used.

Why its worth parallelizing

- This computation will take almost 4 hours to complete and that is only for this 2d model. 3d models will take much longer.
- Our simulation involves 38 million points with 50000 iterations.
- Our parallelization reduces the computation time to under 6 minutes.
- Parallelizing this computation will not just enable to compute these results faster but to perform simulations with larger models otherwise not feasible.

Parallelization Strategy

- **Hybrid:** MPI across cores and nodes, OpenMP across cores within MPI ranks.
- **Domain decomposition:** The grid is easily split into subgrids which can be processed by the individual MPI ranks.
- **Halo Exchange:** Since each point is dependent on the surroundings points in each iteration, bordering elements from one subdomain require the points from the border of the neighbouring subdomain. They are being exchanged each iteration.
- **OpenMP:** To utilize the CPU cores, OpenMP is used to further parallelize the work each rank has been assigned. This can reduce overhead compared to more MPI ranks per compute node since no MPI communication is necessary.

What Was / Wasn't Parallelized, and Why

■ Parallelized:

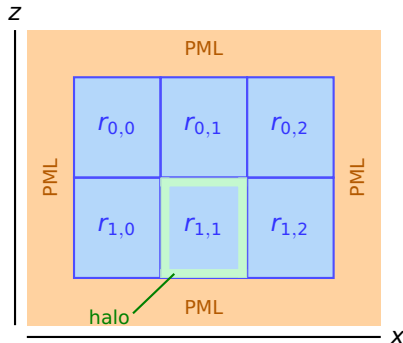
- ▶ **Time-step kernels:** `update_stress` and `update_velocity` use hybrid MPI + OpenMP. Main target of our parallelization.
- ▶ **HDF5 output:** each rank writes its own file independently. Avoids parallel HDF5 or expensive bottleneck if only rank 0 was writing.
- ▶ **Compression:** Blosc is used to parallelize the compression of output data with the OpenMP threads.

■ Not parallelized:

- ▶ **Time loop:** each iteration depends on the previous one, so no parallelism across time steps (Amdahl bottleneck).
- ▶ **One-time setup:** Reading in data, preparing damping stitching together HDF5 files, etc. is cheap relative to 50000 steps and not worth distributing.
- ▶ **Source injection:** single grid point per step, only the owning rank acts, overhead is negligible.

Domain Decomposition

- Global grid split across multiple subdomains.
- Neighbor MPI ranks are automatically found via `MPI_Cart_create`.
- Tile dimensions chosen to minimise halo surface.
- Each rank gets a 1-cell **halo** on all four sides to exchange neighbour data.



Halo Exchange Pattern

Two sweeps per time step:

- **Forward** (before stress update) exchanges v_x, v_z .
- **Backward** (before velocity update) exchanges $\sigma_{xx}, \sigma_{zz}, \sigma_{xz}$.
- Each sweep does **two** Sendrecv calls, one per axis. Each call sends and receives **simultaneously** in opposite directions.
- The stress update only uses the values of the current, the right and the bottom points. The velocity update only uses the values of the current, the left and the top points. This reduces MPI overhead.
- The code checks answers against PROC_NULL to handle domain edges and skips them.

Halo Exchange — Forward

```
def exchange_forward_halos(fields):
    nf = len(fields)
    send_minus = halo_bufs["x_send_minus"][:nf]
    recv_plus = halo_bufs["x_recv_plus"][:nf]
    for k, a in enumerate(fields):
        send_minus[k, :] = a[1:-1, 1]          # pack left column
        cart.Sendrecv(send_minus, dest=x_minus, sendtag=10,
                      recvbuf=recv_plus, source=x_plus, recvtag=10)
    if x_plus != MPI.PROC_NULL:
        for k, a in enumerate(fields):
            a[1:-1, -1] = recv_plus[k, :]      # unpack right ghost
    send_minus = halo_bufs["z_send_minus"][:nf]
    recv_plus = halo_bufs["z_recv_plus"][:nf]
    for k, a in enumerate(fields):
        send_minus[k, :] = a[1, 1:-1]          # pack top row
        cart.Sendrecv(send_minus, dest=z_minus, sendtag=20,
                      recvbuf=recv_plus, source=z_plus, recvtag=20)
    if z_plus != MPI.PROC_NULL:
        for k, a in enumerate(fields):
            a[-1, 1:-1] = recv_plus[k, :]      # unpack bottom ghost
```

Threading Design with OpenMP

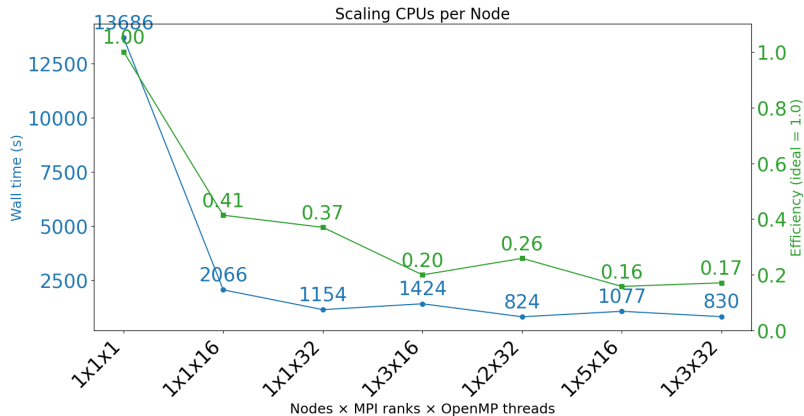
What the pragmas do:

- `omp parallel for schedule(static)`: rows of the local tile are divided evenly across CPU threads before the loop starts.
- `omp simd`: the inner column loop is vectorised, processing 8 floats at once via AVX registers.
- Each grid point reads only its own neighbours and writes only to itself. No race conditions occur, no synchronisation is required.

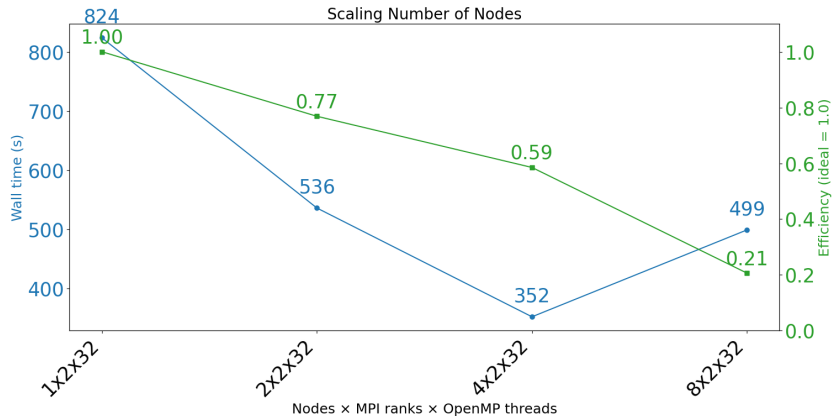
Why `schedule(static)`?

- Every grid point does identical work and there is no branches.
- Static scheduling assigns chunks once upfront, zero runtime overhead.

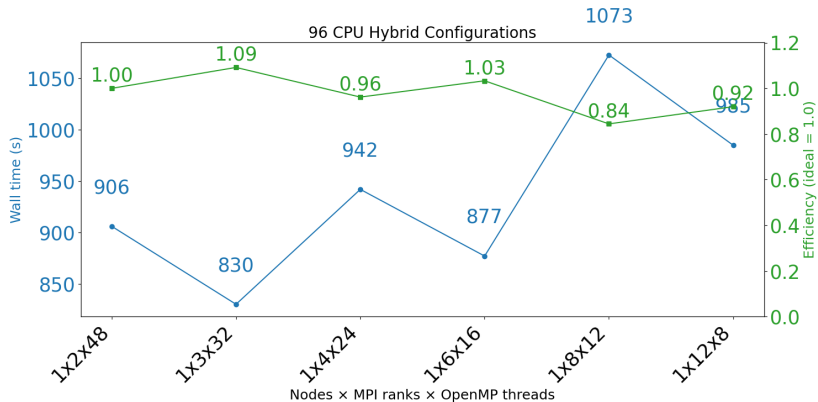
Strong Scaling one Compute Node



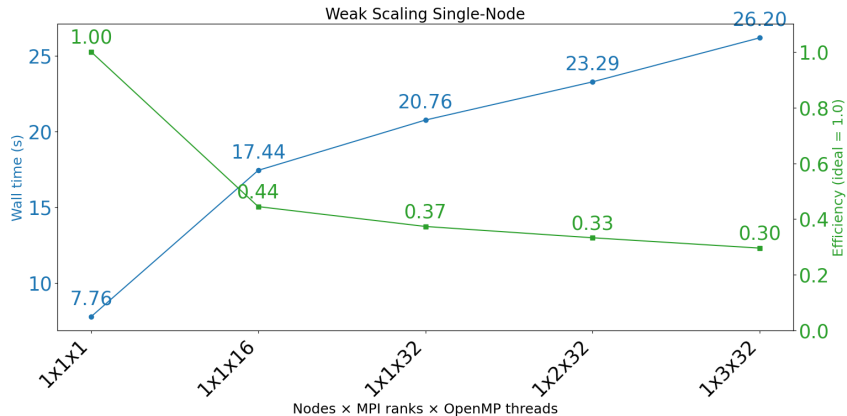
Strong Scaling Multiple Compute Nodes



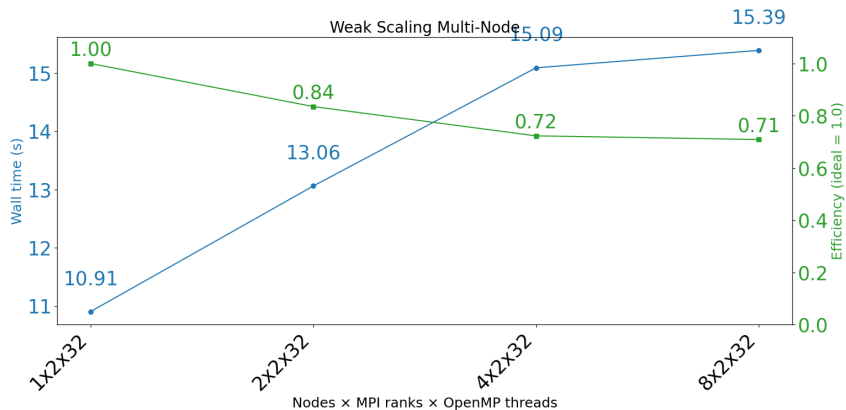
Strong Scaling Various Ranks Per Compute Node with 96 Threads



Weak Scaling One Compute Node



Weak Scaling Multiple Compute Nodes



Performance Depending on Compression Method

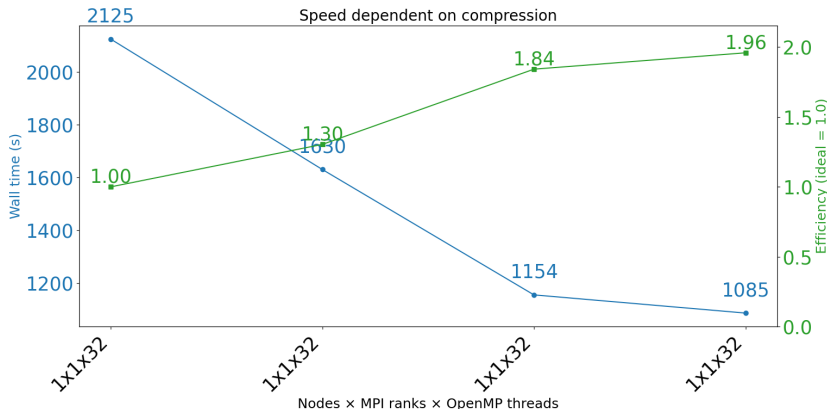


Figure: By compression from left to right: gzip, lz4, Blosc lz4, none

Initial Vampir Analysis

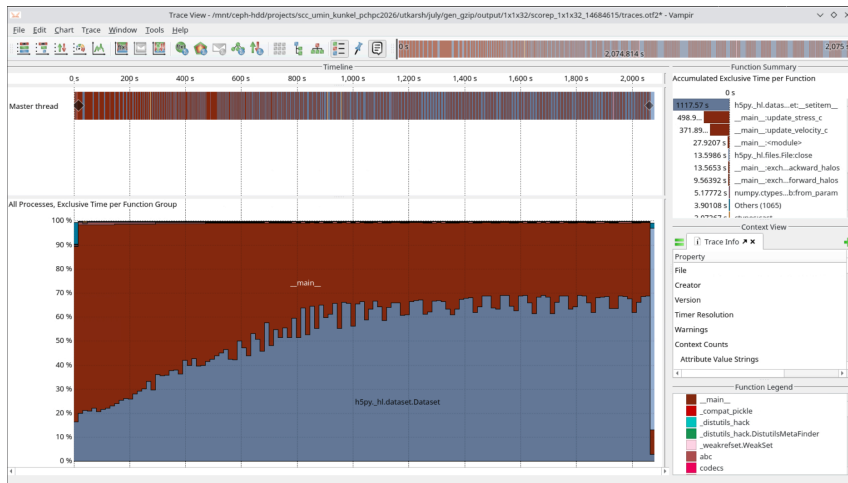


Figure: Vampir Analysis with gzip 1x1x32

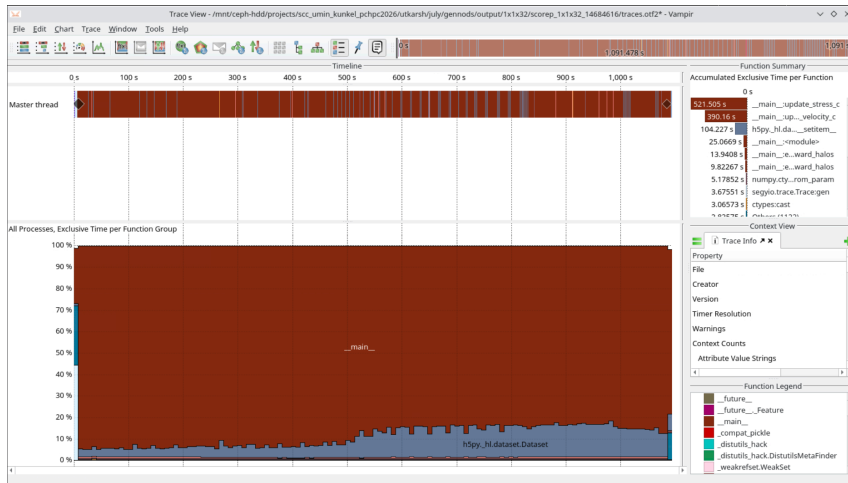
Compression Duration with Gzip



Figure: Compression Time gzip: 2.75s

After changing compression algorithm

using lz4 and blocsc



Compression Duration with lz4 and blosc

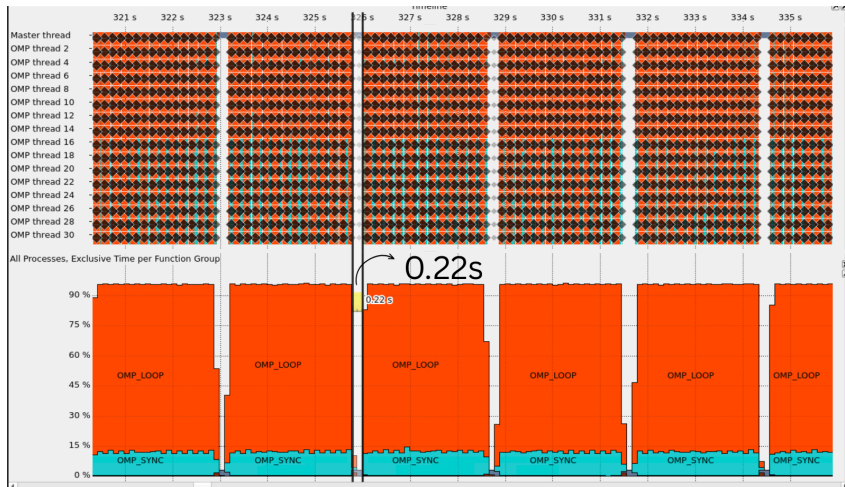


Figure: Compression Time lz4 and blosc: 0.22s

Conclusion

- What we learned about MPI, OpenMP, and cluster computing specifically:
 - ▶ Computations can not just be compute bound but also memory bandwidth bound. Splitting ranks across multiple nodes can therefore be faster.
 - ▶ I/O and compression can be significant bottlenecks.
 - ▶ Vampir is very helpful to detect bottlenecks.
- Goal met: achieving large speedup of runtime by 38 times through parallelization with MPI and OpenMP.
- **Contributions:**
 - ▶ Sequential forward wave simulation on the Marmousi model as the baseline.
 - ▶ Hybrid MPI+OpenMP version.
 - ▶ Full scaling study across configurations including score-p visualization with vampir.

References

Martin, Gary, Robert Wiley, and Kurt Marfurt. “Marmousi2: An elastic upgrade for Marmousi”. In: *The Leading Edge* 25 (Feb. 2006), pp. 156–166. DOI: [10.1190/1.2172306](https://doi.org/10.1190/1.2172306).
Open Data. Accessed: 26.05.2026. 2025. URL: https://wiki.seg.org/wiki/Open_data.