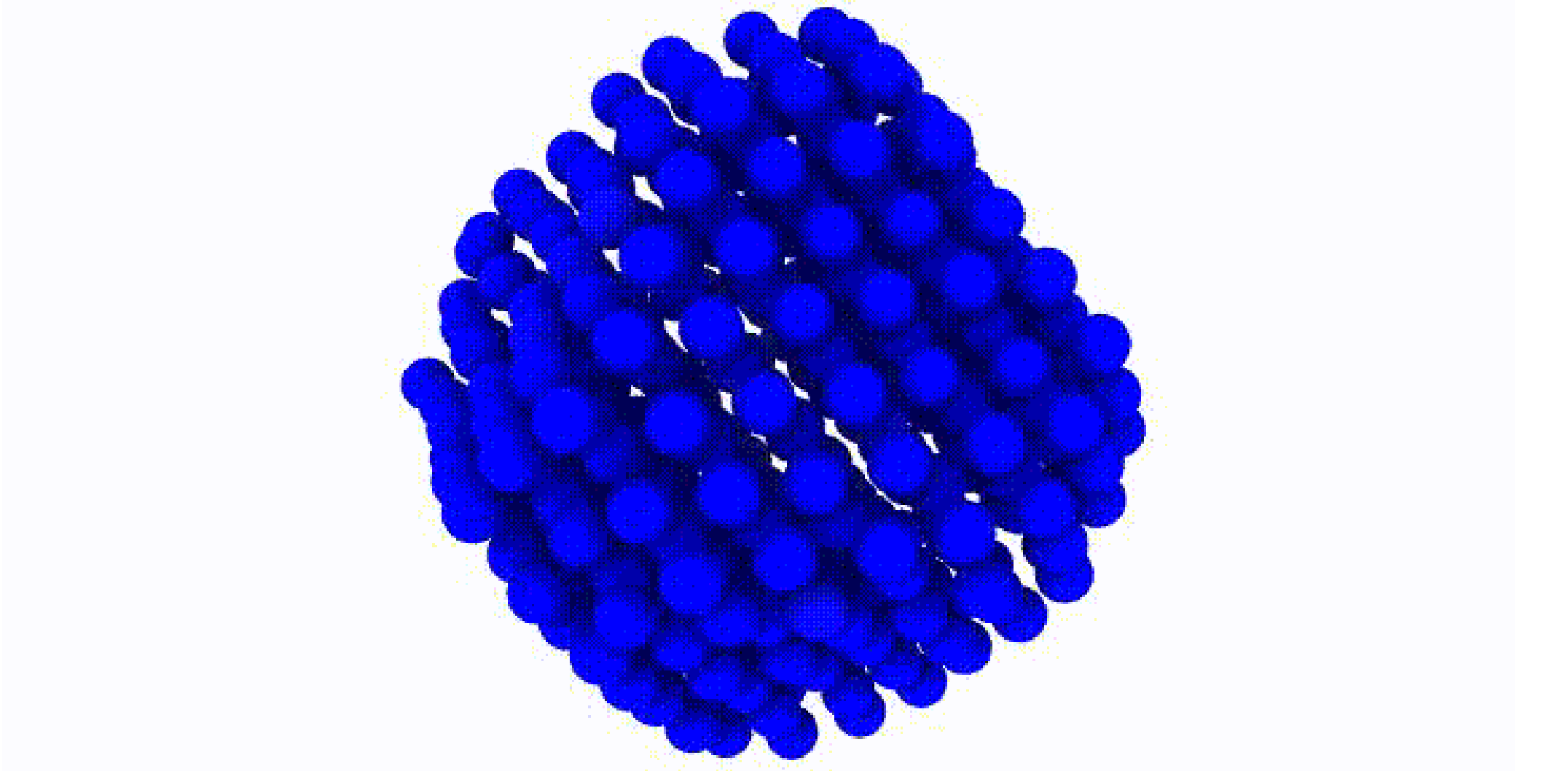
Lindemann Index

Parallel processing of the Lindemann Index useing using MPI and numba

Introduction to the Lindemann Index

- The Lindemann Index is a measure of atomic displacement within a crystal structure.
- It helps in understanding the stability and phase transitions of materials.
- Calculated based on the relative displacements of atoms over a series of frames, e.g. a molecular dynamics simulation.
- I am the maintainer of a Python package that calculates the Lindemann Index (https://github.com/N720720/lindemann)



Lindemann Index

$$q_i = \frac{1}{N-1} \sum_{j(\neq i)} \frac{\sqrt{\langle r_{ij}^2 \rangle_T - \langle r_{ij} \rangle_T^2}}{\langle r_{ij} \rangle_T}$$

and the system-averaged Lindemann index is given by

$$q = \frac{1}{N} \sum_i q_i ,$$

where r_{ij} is the distance between the *i*th and *j*th atoms and $< >_T$ denotes the thermal average at temperature T.

Calculating the Lindemann Index

- Compute pairwise distances between atoms for each frame.
- The Welford algorithm is used to update the mean and variance. This online algorithm was chosen to be able to handle very long simulations without running out of memory.
- Compute Lindemann index, by dividing the square root of the variance by the mean distance and averaging the results.
  (Ratio of the root mean square displacement to the mean distance).

Planning

- Open Development
- Creating a github project for the planned improvements
- Use semantic versioning for the new releases
- Try to document all changes and improvements in the github issues
- Use github actions to have a CI/CD pipeline
- Write unit tests for the new functions to ensure that the results remain the same.
- Publish the new releases to pypi

Current Strategy

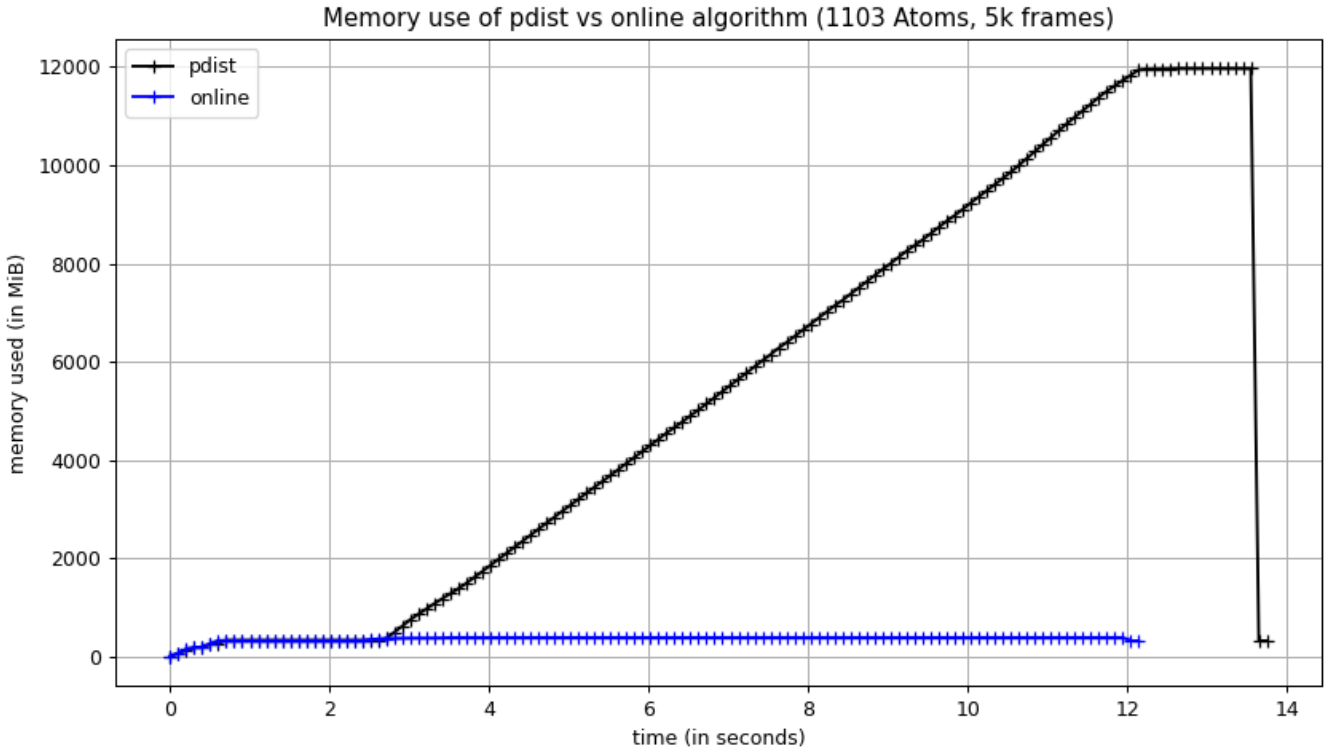
- Using numba to Just In Time compile (JIT) python code
- The complete Molecular Dynamic Simulation is loaded frame by frame
- For each frame, the pairwise distance is calculated
- Then the variance and the mean is updated with the Welford algorithm

Optimizations

- To calculate the pairwise distances between atoms for each image, we could use established libraries to accomplish this task
- Scipy's scipy.spatial.distance.pdist was tested to calculate the pairwise distances
- This would have the advantage that we could use the vectorization of the Scipy vector library

Optimizations

- Unfortunately pdist could not be used, it uses too much memory




```python
frames, atoms, dimension = positions.shape
res = np.zeros((len(positions), dist_len), dtype=np.float32)
for i,position in enumerate(positions):
    res[i] = pdist(position).astype(np.float32)
#The second variant was even worse.
res = np.array([pdist(position).astype(np.float32) for position in positions]).astype(np.float32)
```

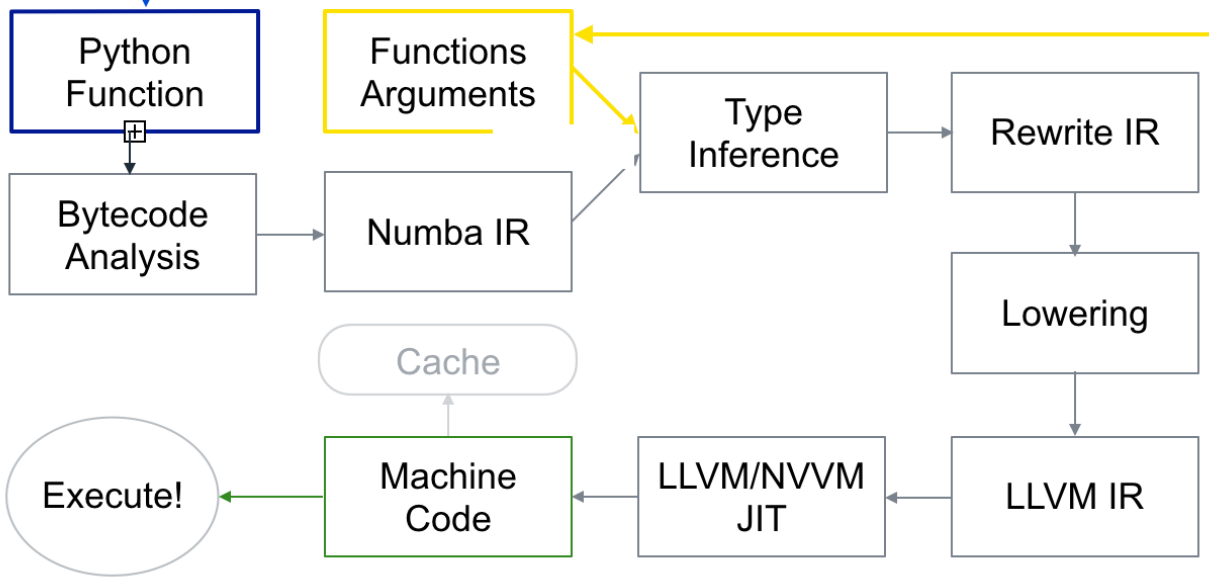
Optimizations

- We can only take functions from numpy and scipy into the jit areas that are supported by numba
- As we are dependent on numba, we have to live with the restrictions here

Numba JIT Compiler

- The code is compiled just in time (jit) with numba

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



Memory Optimizations

- The original implementation of the Lindemann index calculation used a square-form distance matrix.
- This optimization replaces the square-form distance matrix with a distance vector.
- The distance vector requires N*(N-1)/2 storage space, compared to the N^2 space needed for a full distance matrix, where N is the number of atoms.

Optimizations

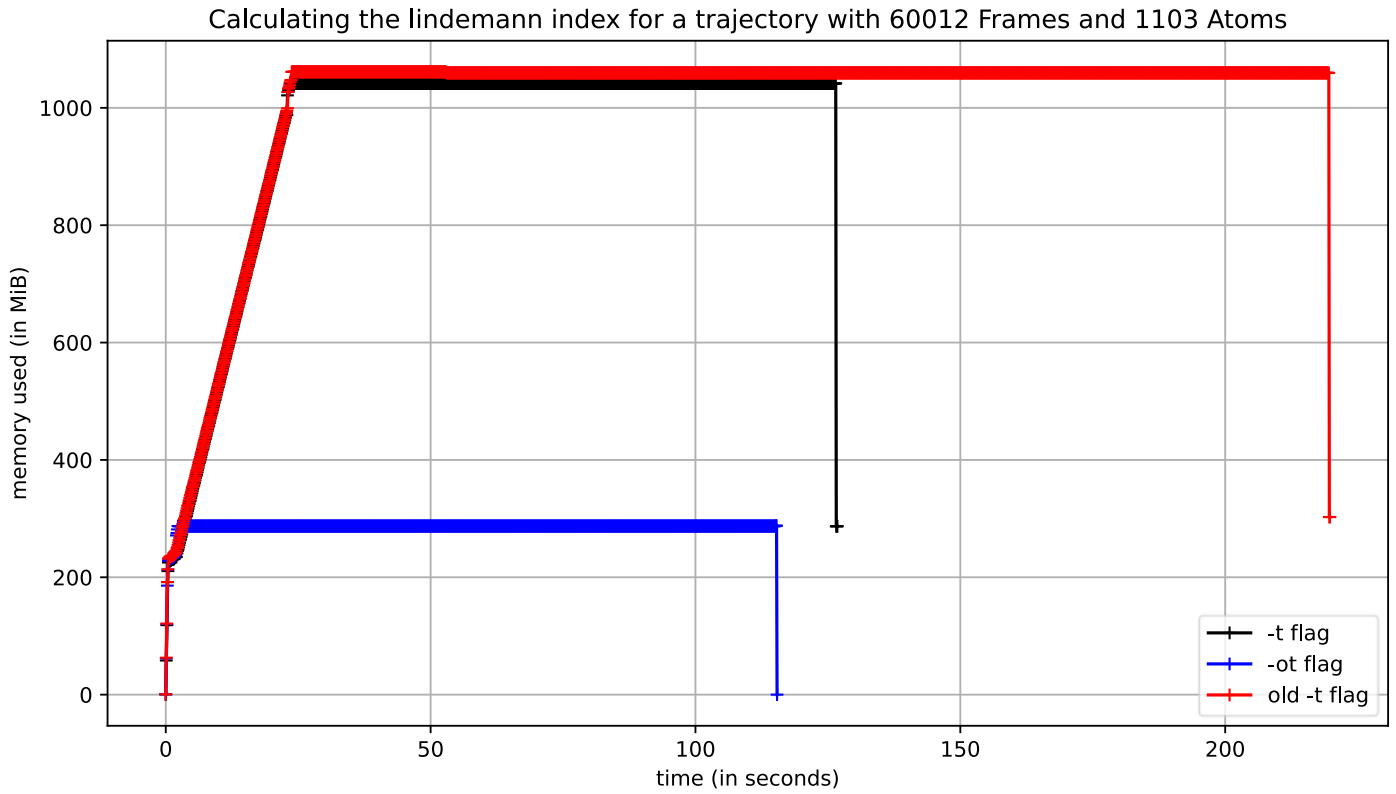
- After further work on the algorithm, I realized that I could merge the loop that calculates the pairwise distance and the the Welford algorithm loop that updates the variance and mean.

```python
@nb.njit(fastmath=True)
def calculate(positions: npt.NDArray[np.float32]) -> np.floating[Any]:
    num_frames: int, num_atoms: int, _: int = positions.shape
    num_distances: int = num_atoms * (num_atoms - 1) // 2

    mean_distances: NDArray[floating[_32Bit]] = np.zeros(num_distances, dtype=np.float32)
    m2_distances: NDArray[floating[_32Bit]] = np.zeros(num_distances, dtype=np.float32)
    for frame in range(num_frames):
        index = 0
        frame_count: int = frame + 1
        for i in range(num_atoms):
            for j in range(i + 1, num_atoms):
                dist = 0.0
                for k in range(3):
                    dist += (positions[frame, i, k] - positions[frame, j, k]) ** 2
                dist: Any = np.sqrt(dist)
                delta: Any = dist - mean_distances[index]
                mean_distances[index] += delta / frame_count
                delta2: Any = dist - mean_distances[index]
                m2_distances[index] += delta * delta2
                index += 1

    return np.mean(np.sqrt(m2_distances / num_frames) / mean_distances)
```

Optimizations on loading the Trajectory

- Loading entire trajectories into memory is often unfeasible for large datasets.
- OVITO's on-demand frame loading reduces memory usage by loading frames as needed.
- Welford's online algorithm updates mean and variance incrementally, allowing frame-by-frame processing without storing all frames in memory.
- We can therefore omit the frames f in the equation f * N * (N-1)/2 for memory usage

Optimizations



Parallel Algorithm

- The existing algorithm is executed serially, limiting performance and scalability.
- The goal is to optimize the existing algorithm by introducing parallelization using Numba's just-in-time (JIT) compilation and parallel processing capabilities (numba prange).
- Laying the foundations for the development of a further mpi4py version

Parallel Algorithm

- Parallel Variance Calculation: Use Welford's algorithm with Numba for numerical stability and performance.
- Chunk-wise Processing: Split frames into chunks for parallel processing (using prange). Compute mean and variance for each chunk independently.

Parallel Algorithm

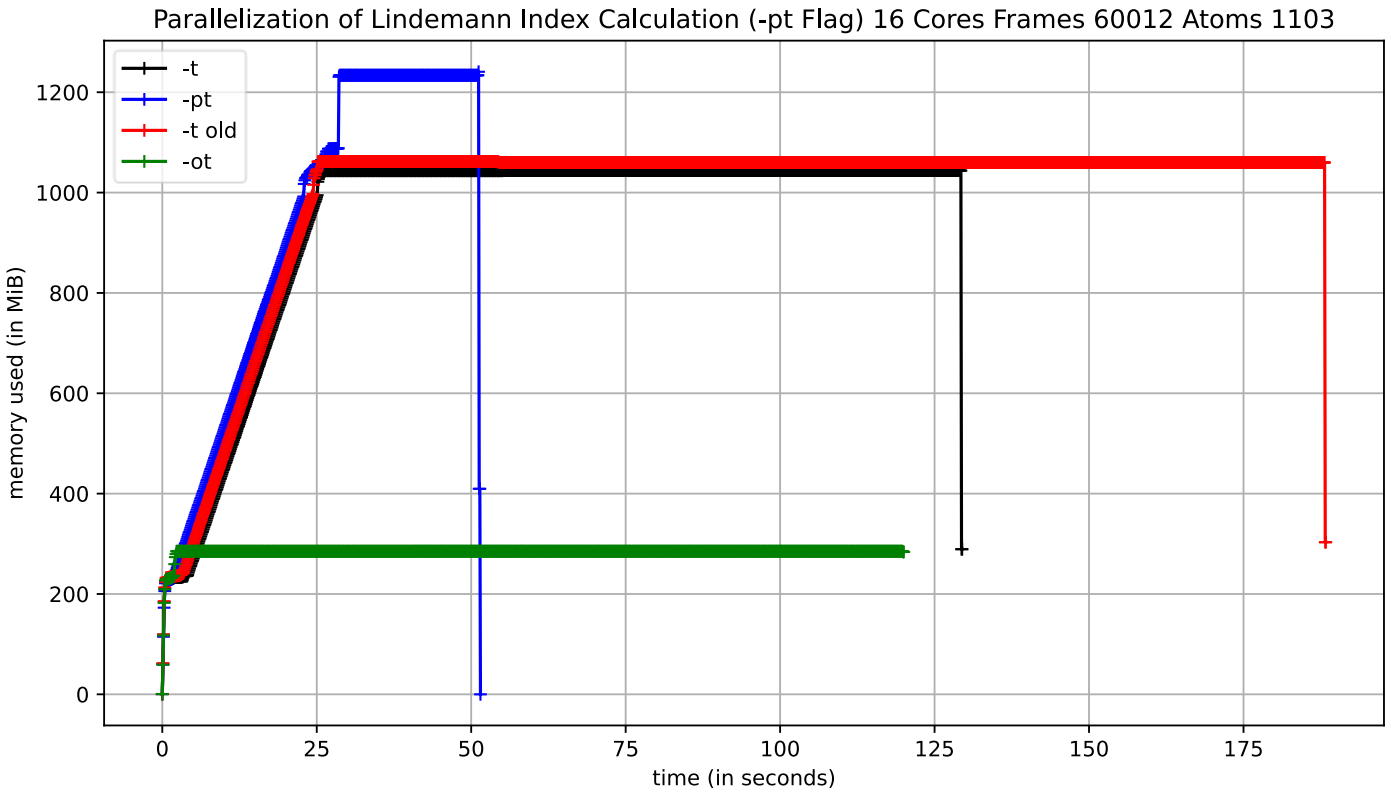
- Combining Results: Aggregate results from all chunks to compute the final Lindemann index efficiently.
- Chan et al.[1] notes that Welford's online algorithm is a special case of an algorithm that works for combining arbitrary sets A and B:

$$
\begin{aligned}
n_{AB} &= n_A + n_B \\
\delta &= \bar{x}_B - \bar{x}_A \\
\bar{x}_{AB} &= \bar{x}_A + \delta \cdot \frac{n_B}{n_{AB}} \\
M_{2,AB} &= M_{2,A} + M_{2,B} + \delta^2 \cdot \frac{n_A n_B}{n_{AB}}
\end{aligned}
$$

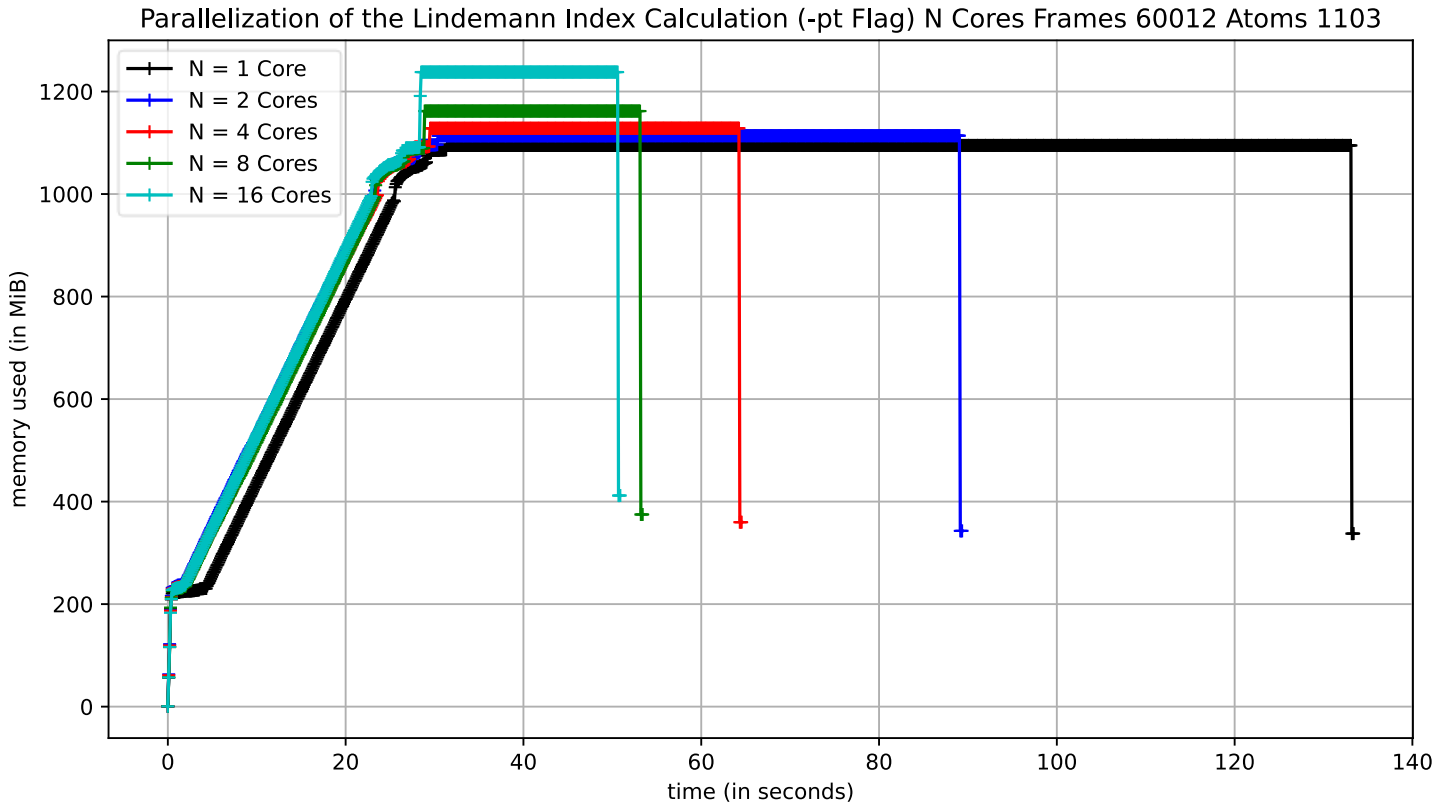
```python
def parallel_variance(n_a, avg_a, M2_a, n_b, avg_b, M2_b):
    n = n_a + n_b
    delta = avg_b - avg_a
    M2 = M2_a + M2_b + delta**2 * n_a * n_b / n
    var_ab = M2 / (n - 1)
    return var_ab
```

[1] Chan et al. *"Updating Formulae and a Pairwise Algorithm for Computing Sample Variances"*

Parallel Algorithm



Parallel Algorithm - CPU Scaling



Parallel Algorithm – With mpy4py

- Ideal for distributed computing across multiple nodes.
- Better scalability and performance in cluster environments.
- Efficiently utilizes multiple nodes. Enhances scalability for large-scale computations. Potential for significant performance improvements.
- Groundwork laid with Numba prange

Parallel Algorithm – With mpy4py

- Mpi4py shows very similar performance to the numba prange version. This is not surprising as both use the omp threading layer.
- Since the scaling of the selected problemset already decreases at 16 Cpu, no attempt has yet been made to distribute the work over several nodes.
- For this purpose, a synthetic problem set must be created that simulates such workloads.

Serial and Parallel Algorithm in C

- Both versions have been developed.
- The shared libraries can be integrated via the python package ctypes.CDLL.
- The serial C version shows better performance than the numba version, approx. 100 seconds vs. 125 seconds

Serial and Parallel Algorithm in C

- It still needs to be explored how the C code can be distributed
- One possibility would be to compile it on a manylinux container in CI/CD to support as many systems as possible and then distribute the shared libraries with the package.
- This would also require the upload to pypi to be integrated into the CI/CD.
- Another option would be to compile the C code on the target system when the package is installed. Here one could use the native arch flag of the compiler.

Utilities and Helpers

- A series of benchmarks were created
- A utility program for calculating memory requirements

Outlook

- Testing of various compilers clang, gcc, icx (godbolt.org)
- Strategy for the distribution of C code
- Creation of documentation to support the user
- Conda package