ETH zürich

MPI Version of GT4Py Code

HPC4WC Project by

Belinda Hotz & Michael Klein &
Vishnu Selvakumar & Yijun Wang

Supervised by

Oliver Fuhrer

Zurich, August 31, 2024

Contents

1 Introduction

In high-performance computing (HPC), efficient parallelization is crucial for exploiting the full potential of modern computing architectures. In this context, Message Passing Interface (MPI) and GridTools for Python (GT4Py) are two essential technologies that enable distributed computing by spreading tasks across multiple processors or nodes to accelerate computations.

This project explores the use of MPI and GT4Py in modern scientific stencil computing, specifically how they improve computational efficiency and scalability. Therefore, we will investigate how the message-passing capabilities of MPI complement the stencil computation framework with GT4Py. With this analysis, we aim to investigate the combined benefits of these two tools in the HPC environment.

1.1 GT4Py

Stencil computations perform mathematical operations to structured grids, updating each grid point based on its neighbours. These computations are computationally expensive, demanding for efficient parallelization. GT4Py is a specialized library designed to unify and optimize those stencil computations, particularly for applications in weather and climate modelling (ETH Zürich, 2014).

A main advantage of GT4Py is its support for multiple hardware backends, enabling the automated parallelization of stencil computations across diverse architectures. The following backends are supported:

- `numpy`: Pure-Python backend
- `gt:cpu_ifirst`: GridTools C11 CPU backend using I-first data ordering
- `gt:cpu_kfirst`: GridTools C11 CPU backend using K-first data ordering
- `gt:gpu`: GridTools backend for CUDA
- `cuda`: CUDA backend with minimally using utilities from GridTools

Furthermore, GT4Py includes tools for managing memory layouts and optimizing data structures based on specific requirements of each backend, ensuring efficient executions of the computation, regardless of the target hardware.

1.2 MPI

MPI is a major Application Program Interface (API) for parallel computing, allowing processes to communicate through message passing in a distributed memory environment Dalcin, 2024. Unlike shared memory systems, MPI operates on systems in which each process has its local memory. Therefore, explicit communication between processes is essential to exchange data. With MPI, processes communicate through a well-defined set of functions and protocols, enabling programs to scale efficiently across multiple compute nodes and making MPI crucial for high-performance computing applications.

Blocking and non-blocking communication models are supported in MPI, each serving different needs for performance and efficiency. In blocking communication, enabled by using `MPI_Send` and `MPI_Recv`, processes must wait until a message has been fully sent or received before they continue

their tasks, ensuring data consistency. However, if processes are waiting for each other, this can lead to idle time. Contrarily, non-blocking communication allows one to initiate data transfers while continuing other tasks simultaneously. `MPI_Isend` and `MPI_Irecv` enable overlapping of computation and communication, significantly improving the overall performance. However, non-blocking communication adds complexity and requires synchronization between processes. Therefore, data dependencies must be handled correctly. Furthermore, all communications must be complete before accessing the transmitted data. MPI provides mechanisms such as barriers to synchronize processes, guaranteeing that all processes reach a specific point before any can proceed.

Additionally, MPI manages data distribution and collection across multiple processes, which is essential for applications that require extensive data manipulation and aggregation across multiple nodes. To efficiently distribute data across processes and collect results, MPI includes operations such as `MPI_Scatter`, distributing data from a single process to all other processes in a communicator, and `MPI_Gather`, collecting data from all processes and assembling it into a single array on the root process. These data movement operations facilitate parallel processing by dividing large datasets into manageable chunks and aggregating the results.

MPI provides the basis for creating efficient and scalable parallel applications. By enabling control over process communication and coordination, MPI allows to optimize applications for a wide range of high-performance computing environments, from small clusters to the world's largest supercomputers.

1.3 Objectives

This project aims to integrate MPI into GT4Py, resulting in a stencil that fully exploits the Piz Daint supercomputer. The main objectives are:

- Implement an integration of MPI into GT4Py for all available backends
- Analyze, optimize and compare the performance for each backend
- Do a weak and strong scaling investigation on Piz Daint for different backends

2 Methods

The stencil program used for the project focuses on implementing a 4th-order diffusion stencil using GT4Py. The key components of the implementation include the definition of diffusion and copy stencils, the management of halo points, and the application of diffusion across a 3D grid. Here's an overview of the main parts:

1. **Stencil definitions:** The `laplacian` function defines a 4th-order finite difference approximation of the Laplacian operator, which is used in the diffusion stencil. The `diffusion_defs` function applies this Laplacian operator to compute the diffusion effect. An advanced version with the two Laplacian operators combined is also used to improve computational efficiency.

2. **Halo management:** The `update_halo` function is responsible for updating the halo regions at the boundaries of the 3D grid. This involves copying data from the interior of the grid to the edges (including corners for left and right edges) to ensure correct boundary conditions.

3. **Applying diffusion:** The `apply_diffusion` function updates halo regions and runs the diffusion calculation, and then swaps input and output fields for iterative processing.

4. **Execution and timing:** The `main` function sets up the grid, initializes fields, and configures the backend for GT4Py execution. It also handles file I/O for saving and plotting the results. The timing of the diffusion process is recorded to evaluate performance.

2.1 Implementation of MPI communication

In this MPI-based implementation, the primary objective is to manage halo exchanges at the boundaries of a 3D grid, focusing on efficient communication and data handling. The key aspects of the implementation include:

1. **Non-blocking communication:** The implementation employs MPI's non-blocking communication functions (`Irecv` and `Isend`) to handle data exchange. Non-blocking communication allows the program to initiate sends and receives without waiting for the completion of these operations, enabling simultaneous communication and computation. The MPI requests created for the receives and the sends are stored in lists (`reqs_tb` for top and bottom edges, and `reqs_lr` for left and right edges).

2. **Synchronization with `wait`:** After initiating non-blocking receives and sends, the code waits for the completion of these operations using `req.wait()`. This ensures that the data has been fully received and processed before unpacking. For the top and bottom edges, the code waits for the completion of the receive operations before updating the halo points. Simultaneously, for the left and right edges, the receive operations wait to be completed before updating the halo regions.

3. **Data packing and unpacking:** Data is packed into buffer arrays for sending and unpacked into already prepared empty buffers after receiving. The implementation creates the data arrays for sending and receiving using `ascontiguousarray` from NumPy. This ensures memory allocation for receiving data and that the data is being sent is correctly formatted and separated from the original grid data. After transmission, the received data is unpacked into the corresponding halo regions of the grid.

By combining contiguous arrays, non-blocking communication, and explicit synchronization with `wait`, this implementation efficiently handles boundary exchanges in a 3D grid while maintaining performance across different computational backends.

Listing 2.1 shows an example of MPI communication in our code.

```
1 l_rcvbuf = gt.storage.from_array(field[0:num_halo, :, :], backend,
     dorigin)
2 r_rcvbuf = gt.storage.from_array(field[-num_halo:, :, :], backend,
     dorigin)
3 reqs_lr = []
4 reqs_lr.append(p.comm().Irecv(l_rcvbuf, source = p.left()))
5 reqs_lr.append(p.comm().Irecv(r_rcvbuf, source = p.right()))
```

Listing 2.1: Part of a Halo Update

2.2 Use of contiguous array

To optimize data transfer and ensure compatibility across different backends, the code uses `np.ascontiguousarray` to consecutively allocate memory buffers for receiving data. A contiguous array is a block of memory where elements are stored sequentially (without any gaps). This layout is crucial for an efficient data access and transfer. When arrays are contiguous, data can be quickly accessed and transferred as a single block, which is essential for MPI communication. Contrarily, non-contiguous arrays, where data elements are scattered across memory, can lead to inefficient memory access and increased communication overhead since MPI must gather scattered elements before sending them, slowing down communication and impacting overall performance.

Furthermore, different backends (such as CPUs or GPUs) may store arrays in different memory layouts, row-wise or column-wise, and without non-contiguous arrays, the seamless transition between the backends is not possible. Non-contiguous arrays can exacerbate these issues by matching the expected data layout, reducing the overhead and making efficient data access and transfer even more difficult. The use of contiguous arrays also ensures that data is consistently and efficiently packaged for transfer, minimizing communication latency and making the code portable across different architectures, whether running on CPUs, GPUs, or other HPC environments.

2.3 Modification of the Laplacian

In order to improve the runtime of the stencil, a modification of the Laplacian, which is used for the diffusion, is implemented.

In our "base" implementation, we call the `laplacian` twice in the diffusion function. In the modified version, we focus on calculating the diffusion with weights for each neighbouring input points (Listing 2.2). By doing so, we hope to optimize the `diffusion_defs`.

```
1     a1 = -alpha
2     a2 = -2 * alpha
3     a8 = 8 * alpha
4     a20= 1 - 20 * alpha
5
6     out_field = (
7         a1  * in_field[ 0, -2, 0] + a2  * in_field[-1, -1, 0] +
8         a8  * in_field[ 0, -1, 0] + a2  * in_field[ 1, -1, 0] +
9         a1  * in_field[-2,  0, 0] + a8  * in_field[-1,  0, 0] +
10        a20 * in_field[ 0,  0, 0] + a8  * in_field[ 1,  0, 0] +
11        a1  * in_field[ 2,  0, 0] + a2  * in_field[-1,  1, 0] +
12        a8  * in_field[ 0,  1, 0] + a2  * in_field[ 1,  1, 0] +
13        a1  * in_field[ 0,  2, 0] )
```

Listing 2.2: Modified diffusion

3 Results and discussion

The following chapter provides a comprehensive analysis of the code, highlighting key findings and their implications as well as discussing potential limitations and the broader context.

3.1 Testing the MPI-integrated GT4Py implementation

First, the focus lies on the validation of the code, ensuring that our updated version works correctly. Therefore, we verify the input and output of the MPI-integrated GT4Py implementation (Fig. 3.1), applied on a grid with the dimensions nx=1028, ny=1028, nz=128 for 256 iterations. It demonstrates that the updated code works as intended.

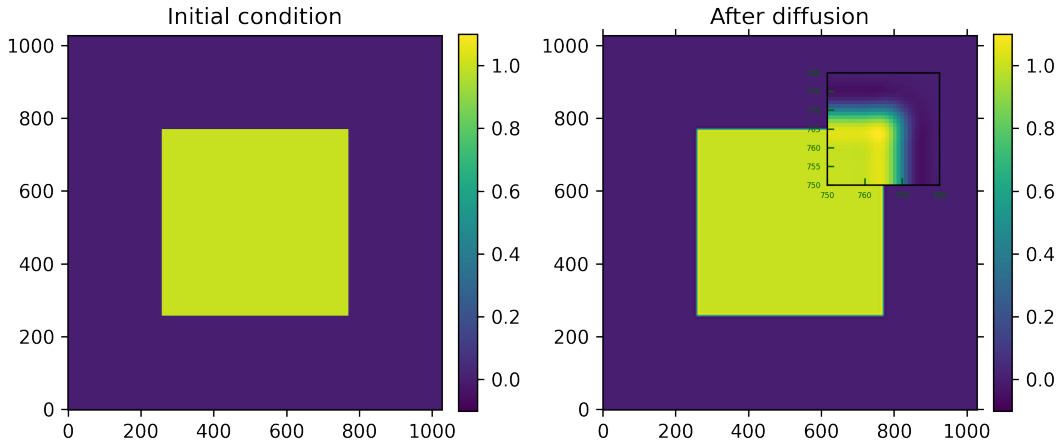

Figure 3.1: Fields before and after diffusion using GT4Py + MPI, for 1028 x 1028 x 128 grid size and 256 iterations.

3.2 Strong scaling of the MPI-integrated GT4Py implementation

This section explores the capabilities of strong scaling of our computational approach, assessing the effect of increasing the number of processors on the runtime for a fixed workload.

3.2.1 Comparison to MPI version

To analyse the performance of the MPI-integrated GT4Py stencil code, we first evaluate the performance of the `numpy` backend compared to the original MPI stencil using a single socket. Therefore, a grid size of 1028 x 1028 x 128 over 256 iterations was chosen.

The MPI-integrated GT4Py implementation shows a decrease in runtime as the number of cores increases, particularly up to about ten cores (Fig. 3.2). Afterwards, the runtime stagnated. This behaviour likely comes from the memory bandwidth of the system, being the limiting factor of the runtime as the computation is memory-bound. Since we only use a single socket, the gain in reducing runtime by adding more cores is constrained by the available memory bandwidth, explaining why the runtime does not scale with the number of cores.

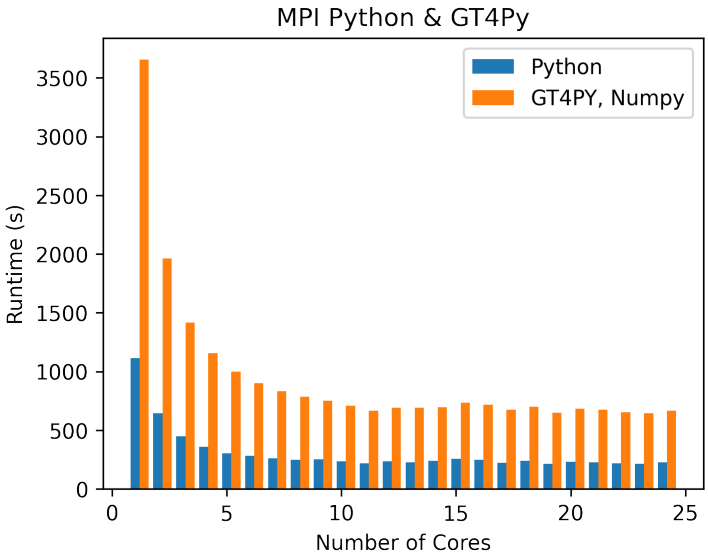

Figure 3.2: Comparison of the runtime and MPI performance of a Python implementation (blue) with a MPI-integrated GT4Py implementation (orange) using the `numpy` backend across different numbers of cores for 1028 x 1028 x 128 grid size and 256 iterations on a single socket.

Despite being specifically designed for parallel computations, the MPI-integrated GT4Py implementation consistently runs slower (∼3 times higher) compared to the MPI Python implementation. This performance discrepancy is likely due to the internal limitations of the `numpy` backend used in GT4Py. While `numpy` is efficient for general numerical computations, it is not optimized for highly parallelized tasks. Consequently, the MPI-integrated GT4Py version fails to leverage the full potential of the parallelization, resulting in slower runtime compared to the standard Python implementation. This performance gap highlights the importance of a more optimized backend in GT4Py to handle large-scale parallel computations efficiently, which will be discussed in the next step.

3.2.2 Comparison of different backends

Comparing the CPU backends of the MPI-integrated GT4Py implementation to the previous performance using the `numpy` backend, the runtime shows a noteworthy improvement by using the `gt:cpu_kfirst` and `gt:cpu_ifirst` backends (Fig. 3.3). This improvement highlights the benefit of using backends specifically designed for parallel CPU execution in GT4Py, because they optimize the parallel computations. The difference in runtime between these CPU backends, with `gt:cpu_kfirst` consistently showing slightly longer runtimes than `gt:cpu_ifirst` suggests that the ordering of loop execution (i-first vs. k-first) can have a notable impact on performance, with `gt:cpu_ifirst` being more efficient in this specific scenario. The less optimal configuration of `gt:cpu_kfirst` could be due to less efficient memory access patterns, leading to higher runtimes. Nevertheless, both CPU backends exhibit increasing runtimes as the number of cores increases, reflecting the differences in how these backends manage parallel computation.

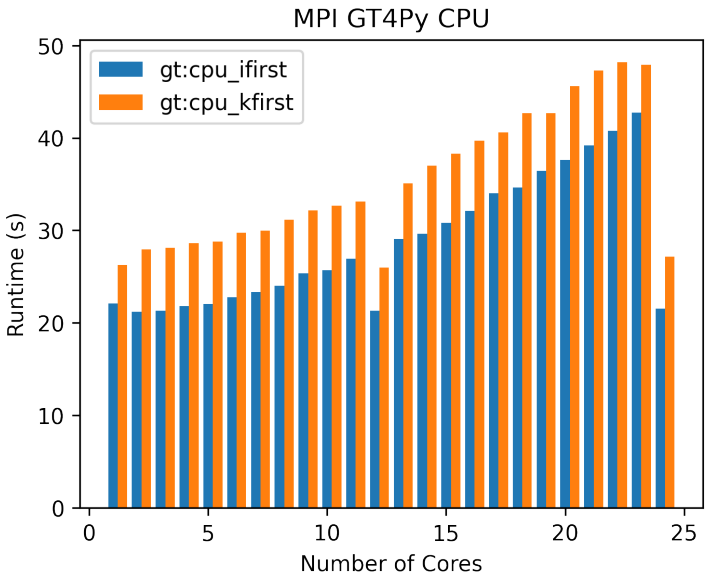

Figure 3.3: Comparison of runtime and MPI performance of two GT4Py CPU backends, `gt:cpu_ifirst` (blue) and `gt:cpu_kfirst` (orange), across different numbers of cores for 1028 x 1028 x 128 grid size and 256 iterations on a single socket.

For the `numpy` backend, the runtime decreases significantly as the number of cores increases. This is likely due to the fact that `numpy`, being primarily a library optimized for single-threaded performance, benefits from parallel execution when combined with MPI. The workload is distributed among multiple cores, reducing the runtime as more resources are allocated to the task. However, the `numpy` backend is not inherently designed for parallel computing in the same way the GT4Py CPU backends are, which is why its performance is substantially lower even when using multiple cores.

In contrast, the `gt:cpu_ifirst` and `gt:cpu_kfirst` backends exhibit increasing runtimes as the number of cores increases. This counter intuitive trend can be attributed to the overhead associated with managing parallel execution across many cores, particularly in the context of these specific backends. By increasing the number of cores, the communication overhead, synchronization, and data transfer between cores increase, leading to diminished returns in performance and eventually an increase in overall runtime. This phenomenon is particularly evident in applications where the parallelization strategy is not perfectly balanced, or where the task size does not scale efficiently with the number of cores. There are two significant dips in the runtime with 12 and 24 cores. There are two significant dips in the runtime with 12 and 24 cores, where it remains unclear why they occur.

3.2.3 Comparison to GT4Py implementation without MPI

To quantify the gain in performance by implementing MPI into the GT4Py code, we compare the runtime of a plain GT4Py and MPI-integrated GT4Py implementation. We see that the runtime of the plain GT4Py implementation increases with increasing cores (Fig. 3.4) with lower runtimes for `gt:cpu_ifirst` than for `gt:cpu_kfirst`. Typically, in non-MPI implementation, the increasing number of cores often leads to increased communication overhead due to the exchanged amount of data between the cores. Therefore, the overhead can become a bottleneck, resulting in an observable increase in runtime with an increasing number of cores. Furthermore,

the load between the cores can be unevenly distributed across the cores, causing delays and increasing the total runtime. Moreover, without MPI, each core may have to handle larger portions of the data in the memories, leading to inefficiencies in memory access.

The MPI-integrated GT4Py implementation shows a much smaller increase in the runtime, indicating better scalability. In addition, MPI-integrated GT4Py implementation outperforms the plain GT4Py implementation, regardless of whether `gt:cpu_ifirst` or `gt:cpu_kfirst` is used as a backend. The better performance could be due to less parallel communication overhead, better load balancing between the cores or better memory management. We can conclude that implementing MPI into GT4Py optimizes the performance.

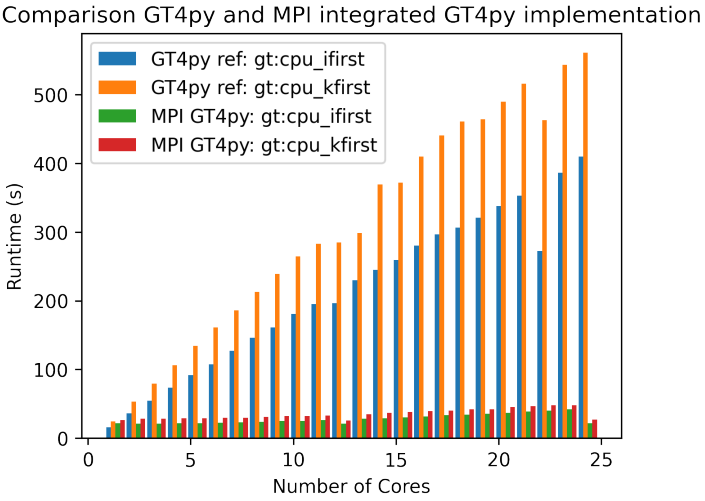

Figure 3.4: Comparison of runtime GT4Py reference implementation and MPI-integrated GT4Py implementation of two CPU backends, `gt:cpu_ifirst` and `gt:cpu_kfirst`, across different numbers of cores for 1028 x 1028 x 128 grid size and 256 iterations on a single socket.

3.2.4 Modification of the Laplacian

An additional speed-up is observable by changing the computation of the Laplacian. This section focuses on finding the best Laplacian version for the MPI-integrated GT4Py stencil to minimize the runtime. In our "base" implementation, we call the Laplacian twice in the diffusion function. In the modified version, we focus on calculating weights for each of the neighbouring input grid points (Listing 2.2).

The modification of the Laplacian outperforms our "base" MPI-integrated GT4Py implementation, indicating that the modification is more efficient (Fig. 3.5). In both implementations, the `gt:cpu_ifirst` backend shows better performance than the `gt:cpu_kfirst` backend, suggesting that `gt:cpu_ifirst` is more optimal for both implementations, likely due to better memory access patterns. As the number of cores increases, the runtime for both implementations shows some variability. However, the Modified Laplacian implementation consistently maintains lower runtimes, indicating that the Modified Laplacian scales better with the number of cores, benefiting more from parallelization.

The better performance could be due to a more efficient algorithm, a reduced computational load or a better memory access pattern. The base version applies twice the Laplacian in a func-

tion. Therefore, applying modified Laplacian might reduce the computational load or redundant calculations or better leverage data locality.

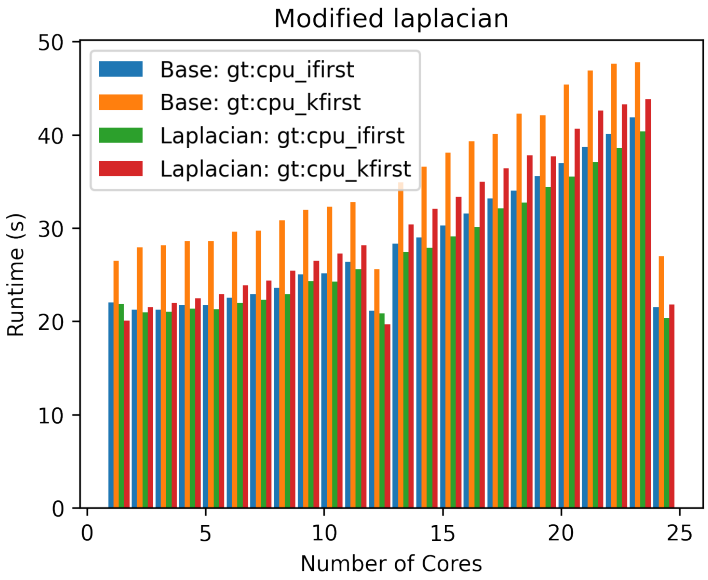

Figure 3.5: Comparison of runtime and MPI performance of two GT4Py CPU backends, `gt:cpu_ifirst` (blue) and `gt:cpu_kfirst` (orange) for our "base" and modified Laplacian implementation, across different numbers of cores for 1028 x 1028 x 128 grid size and 256 iterations on a single socket.

3.3 Weak scaling of the MPI-integrated GT4Py implementation

Here, we evaluate the weak scaling performance, analyzing whether the computation time remains consistent as the problem size grows in proportion to the number of processors. Fig.3.6 illustrates the weak scaling of the code compiled with different backends.

In an ideal weak scaling scenario, as illustrated by the red dashed line in the subplots in Fig. 3.6, the runtime would remain constant regardless of the number of nodes, indicating an evenly distributed workload across the nodes, with minimal overhead and no bottlenecks in communication or resource access.

All backends of our MPI-integrated GT4Py implementation show an increase in the runtime with increasing node count. However, there is a large variation in how the backends cope with the weak scaling. While some backends maintain nearly constant runtimes, others show noticeable increases, revealing their limitations in handling larger distributed computations efficiently. As the problem size increases proportionally with the node count, the overhead associated with inter-node communication, synchronization, and memory handling becomes substantial. This leads to a marked increase in runtime, which deviates significantly from the ideal scaling line. The overhead becomes more apparent with larger node counts, as the cost of data movement and communication between nodes outweighs the benefits of increased computational resources.

When comparing different backends of our implementation, the `cuda` implementation stands out with its near-ideal weak scaling performance. Using GPU acceleration allows the `cuda` backend to handle the increased computational load effectively, keeping the runtimes almost constant across varying node counts. This efficiency is likely due to the high parallelism and advanced memory management capabilities of GPUs. In contrast, the `numpy` backend exhibits the poorest

scaling, with a significant increase in runtimes as the node count grows, likely due to the limited parallel processing capabilities of `numpy` and the overhead associated with managing distributed data across nodes. Contrarily, the `gt:cpu_ifirst` and `gt:cpu_kfirst` show a moderate increase in the runtime, suggesting a more efficient handling of parallelizations. However, these backends still encounter some overhead challenges, possibly from inter-thread communication and memory access inefficiencies.

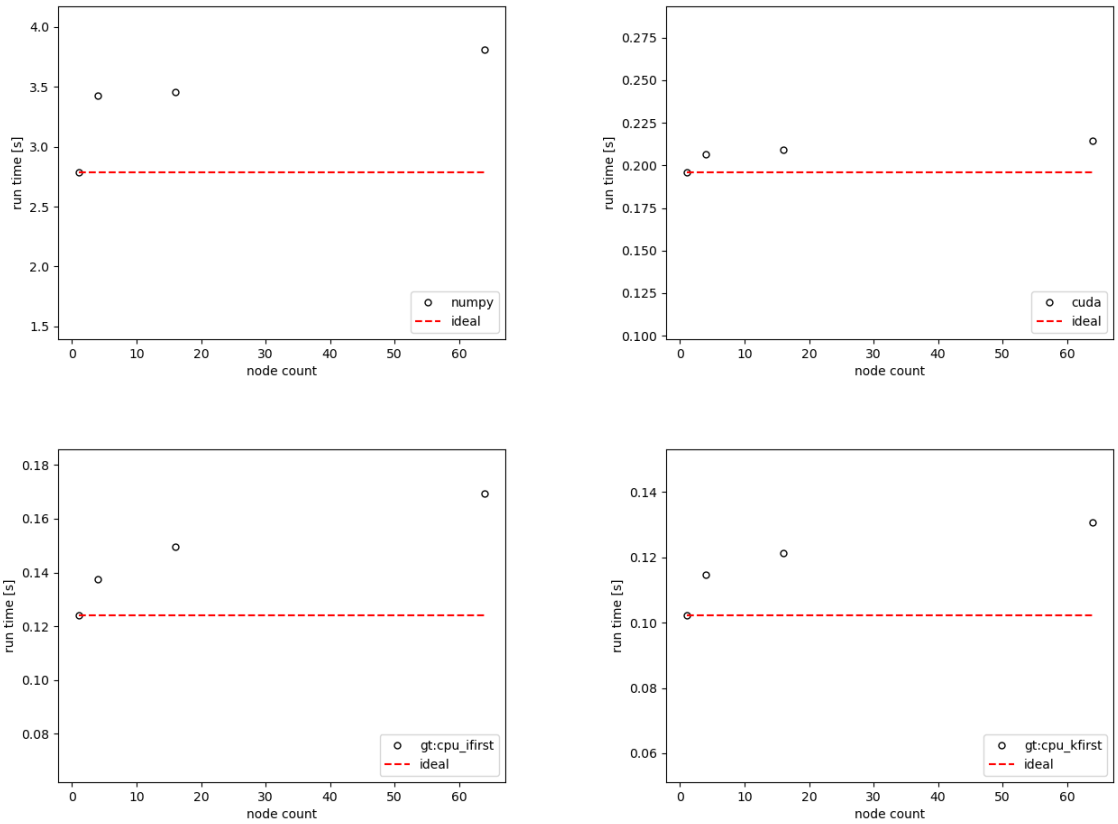

Figure 3.6: Weak scaling of the code compiled with different backends: problem size $n_x \times n_y \in [64 \times 64,\ 128 \times 128,\ 256 \times 256,\ 512 \times 512]$ with a fixed $n_z = 64$ and 128 iterations; node count $\in [1,\ 4,\ 16,\ 64]$. Each node has 12 OpenMP threads.

4 Conclusion and limitations

In this project, a hybrid program utilizing both MPI and GT4Py to complete stencil computations on a structured grid was created. Also, a new version of the Laplacian was implemented. This was then tested for different backends and also compared to the original MPI and GT4Py versions by analyzing the strong and weak scaling of the different programs.

The MPI-integrated GT4Py implementation using the `numpy` backend showed around a 3x slowdown from the MPI Python implementation, which can be attributed to the non-optimized nature of `numpy` in regards to parallel computations. Even though the code showed this slowdown, the usage of different backends resulted in more promising results. Both, the `gt:cpu_kfirst` and `gt:cpu_ifirst` backends showed speedups compared to `numpy`, as they are much more optimized regarding parallelization.

Regarding the strong scaling, `numpy` showed a reduction in runtime with more cores, while both the CPU backends showed a slowdown. This is due to how the backends are optimized for work on single cores, as the CPU backends rely on multi-threading, while `numpy` is much more focused on a single thread. This leads to `numpy` profiting much more from using multiple cores, due to among other things, much lower overhead. This slowdown in the CPU backends however still does not make `numpy` a better choice concerning computation speed.

When comparing the MPI-integrated GT4Py implementation with a plain GT4Py implementation in strong scaling, the results indicated that integrating MPI improved performance. While the plain GT4Py implementation exhibited increased runtimes with more cores, likely due to increased communication overhead and uneven load distribution, the MPI-integrated version scaled better and showed smaller increases in runtime. These findings highlight that MPI integration offers better load balancing, reduced communication overhead, and optimized memory management, enhancing overall scalability and performance.

Introducing a modified Laplacian further improved the performance of the MPI-integrated GT4Py implementation, suggesting that optimized algorithms and efficient memory access patterns are critical for achieving high performance in parallel computing environments.

The weak scaling analysis revealed that the `cuda` backend offers the best performance for large-scale 2D stencil computations, maintaining nearly constant run times due to its efficient use of GPU resources. The GT4Py CPU backends (`gt:cpu_ifirst` and `gt:cpu_kfirst`) provide moderate scaling, outperforming `numpy`, which struggles with increased node counts due to its limited parallel capabilities. These results highlight the importance of choosing a suitable backend to achieve efficient scaling and optimize performance based on the available computational resources.

The work discussed in this project has a few notable shortcomings. First, most of the tests were conducted on a single node, which is not always representative of real-world scenarios where multi-node configurations are often used, potentially affecting the scaling results. Moreover, the problem sizes chosen for testing were relatively similar, which could skew the results in favour of a backend that performs well with that specific level of complexity, possibly overlooking how other backends might perform with more varied problem sizes. GPU-based backends show a communication bottleneck when transferring data to the GPU, leading to higher completion times, which might not accurately reflect the performance of systems with more efficient communication channels. Lastly, the absence of detailed profiling information, such as CPU and GPU utilization, memory access patterns, and cache usage, limits the depth of analysis and understanding of the specific performance bottlenecks. Future work could include these profiling metrics to further optimize and enhance the performance.

5 References

Dalcin, L. (2024). *Mpi for python*. Retrieved August 12, 2024, from https://mpi4py.readthedocs.io/en/stable/#mpi-for-python

ETH Zürich. (2014). *Gt4py: Gridtools for python*. Retrieved August 12, 2024, from https://gridtools.github.io/gt4py/latest/index.html