Exploring advanced visualization of MPI parallel programs

A Framework for High-Performance Computing Insight

Jean-Yves Verhaeghe
Supervisors: Prof. Gerhard Wellein and Dr. Georg Hager
Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg

30.06.2025

Contents

Abstract

The performance analysis of parallel programs, particularly those using the Message Passing Interface (MPI), typically relies on trace data that is challenging to interpret in textual or 2D visualizations.

This thesis proposes a novel approach that leverages the 3D modeling and animation of Blender to visualize MPI trace data in a three-dimensional environment within a video. By mapping each process rank, and communication events onto 3D figures, this project enables intuitive exploration of program behavior, bottlenecks, and communication patterns.

The framework developed offers automated translation of trace data into Blender scenes. This thesis discusses the design, implementation, and evaluation of this visualization pipeline and presents use cases that highlight its potential to enhance understanding of MPI-based parallel program performance.

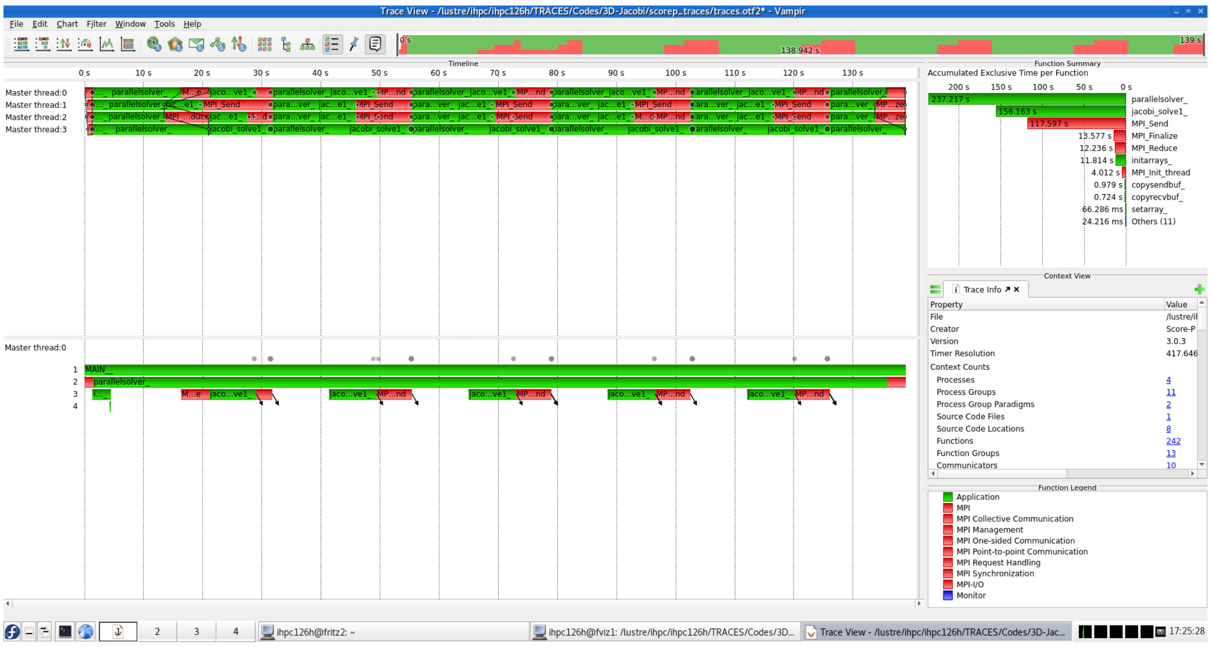
To this end, it uses a real-world scientific application, the Jacobi algorithm, run on our Fritz HPC cluster, and generates idle waves to create a typical use case where this tool could bring an interesting added value. This base setup is investigated through our innovative visualization, which brings both better and faster understandability, shows subtle run details hard to grasp from usual tools, and adapts to higher-dimensional and other complex domains, eventually giving us valuable added insight into the fine details of each run.

Background

1.1 Existing Trace Visualization Tools

Trace visualization tools are essential for understanding and analyzing the performance of parallel and distributed applications. These tools allow developers and researchers to examine trace data generated by performance profiling tools, providing valuable insights into how programs execute across multiple processors or nodes. By visualizing events like function calls, communication patterns, and resource utilization, trace visualization tools help pinpoint inefficiencies, bottlenecks, and areas for optimization in complex parallel systems.

One of the leading tools in this domain, Vampir, is a powerful visualization tool designed for high-performance computing (HPC) applications. It takes trace data generated by profiling tools like Score-P and presents it into interactive, easy-to-interpret visualizations. In particular, the timeline view feature displays a list of all ranks, color-coding events depending on their state of execution or communication along the time, helping users identify performance issues such as load imbalances, excessive communication overhead, and inefficient execution patterns.



Property	Value
File	/lustre/il
Creator	Score-P
Version	3.0.3
Timer Resolution	417.646
Context Counts	
Processes	4
Process Groups	11
Process Group Paradigms	2
Source Code Files	1
Source Code Locations	8
Functions	242
Function Groups	13
Communicators	10

Figure 1.1: Vampir display

1.2 Motivation and Visualization Challenges

Current 2D tools provide timelines and message flow diagrams but often become unreadable as the number of processes increases. While Vampir is a powerful and widely used tool for visualizing MPI trace data, it becomes increasingly difficult to interpret trace information effectively when the data spans more than one dimension. Vampir primarily presents information using 2D timelines—typically mapping time on the horizontal axis and process ranks or threads on the vertical axis.

Another problem is the display of a significant amount of ranks, typically over 100, which is easily attained and can take orders of magnitude more. Also, simultaneous communications between many ranks can create dense, intersecting messages that obscure timing relationships and communication patterns. As a result, users may struggle to extract meaningful insights from large or complex datasets, particularly in massively parallel applications where multidimensional behavior is the norm. This limitation motivates the exploration of alternative visualization approaches, such as 3D representations, to better capture and present the full richness of the trace data. An interesting previous attempt to display computations in 3D is available in the references[13]. For those cases involving

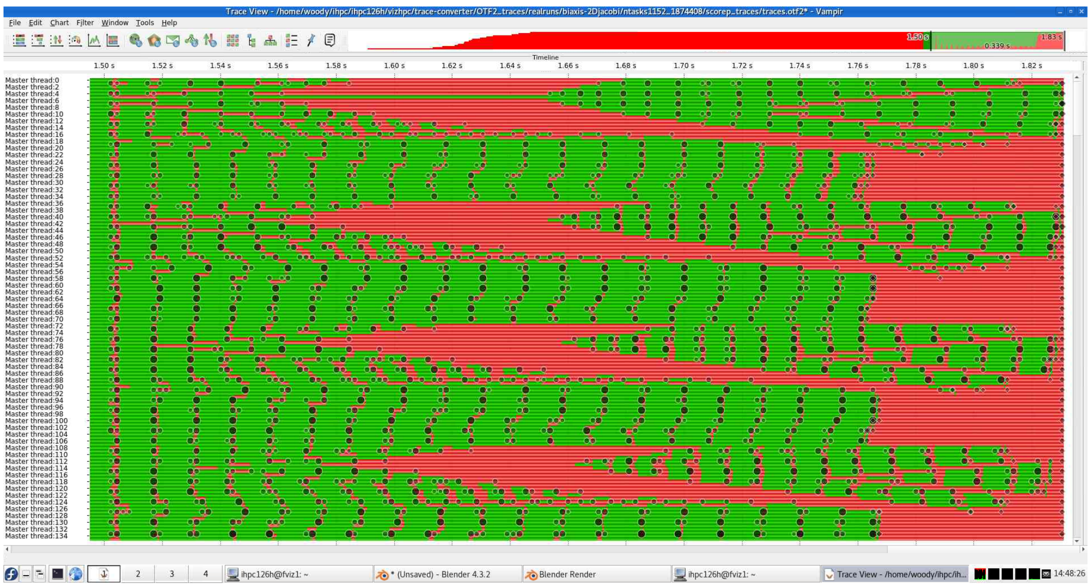

Trace View - /home/woody/ihpc/ihpc126h/vizhpc/trace-converter/OTF2_traces/realruns/biaxis-2Djacobi/ntasks1152_1874408/scorep_traces/traces.otf2* - Vampir
File Edit Chart Filter Window Tools Help
Timeline
1.50 s
1.52 s
1.54 s
1.56 s
1.58 s
1.60 s
1.62 s
1.64 s
1.66 s
1.68 s
1.70 s
1.72 s
1.74 s
1.76 s
1.78 s
1.80 s
1.82 s
Master thread:0
Master thread:134
Blender Render
14:48:26

Figure 1.2: Vampir display on a more than 100 ranks with 2D dimensionality

such high-dimensional communication patterns, and in order to understand them, this thesis proposes a novel approach using intuitive 3D representations of such data and outputting it to a video, facilitating deeper insights.

1.3 Design Objectives and Graphic Framework Choice

The necessary features of our desired tool are to:

- Translate MPI trace data into a 3D spatial representation.
- Display each rank in a fitting way with regard to the source underlying domain and the communication patterns it involves.

- Generate automatically a 3D scene, to be converted into an output video with a script that we keep separate from our extraction script for modularity.
- Explore, test and select visually intuitive outputs for HPC users and researchers.

Candidate frameworks - Pros and Cons

We considered three possible frameworks to implement our visualization, namely `Manim`, `Javascript` and `Blender`.

`Manim` was the first to be excluded, due to its higher complexity, and steeper learning curve. Visualizing the result is less interactive and more constrained, since everything needs to be coded through dedicated functions and rendered before being even able to view the output. The other two offer simple mouse gesture to quickly check what's the best angle and how it looks.

`Javascript` was a very good candidate. It offers many interesting frameworks, works from a simple browser, excluding the need to install additional software, and is by far the most versatile. One example of that is the possibility to display information when hovering objects, which allows for interesting additions. We considered `D3.js`, and `Three.js`, which has the perk to handle 3D glasses.

However, `Blender` seemed like the most balanced decision, with straightforward and user-friendly handling by either coding with a powerful Python API or dedicated quick buttons on the GUI, a long and strong development with a very active community, as well as extensive documentation. Furthermore, Blender specializes in rendering videos, so much so that it can even be used as a standalone video editing tool, which was the desired output, while still having the possibility to interact within Blender with the 3D scene, for example by changing interactively its zoom and angle while it is playing. It is the tool we selected this thesis' visualization objectives.

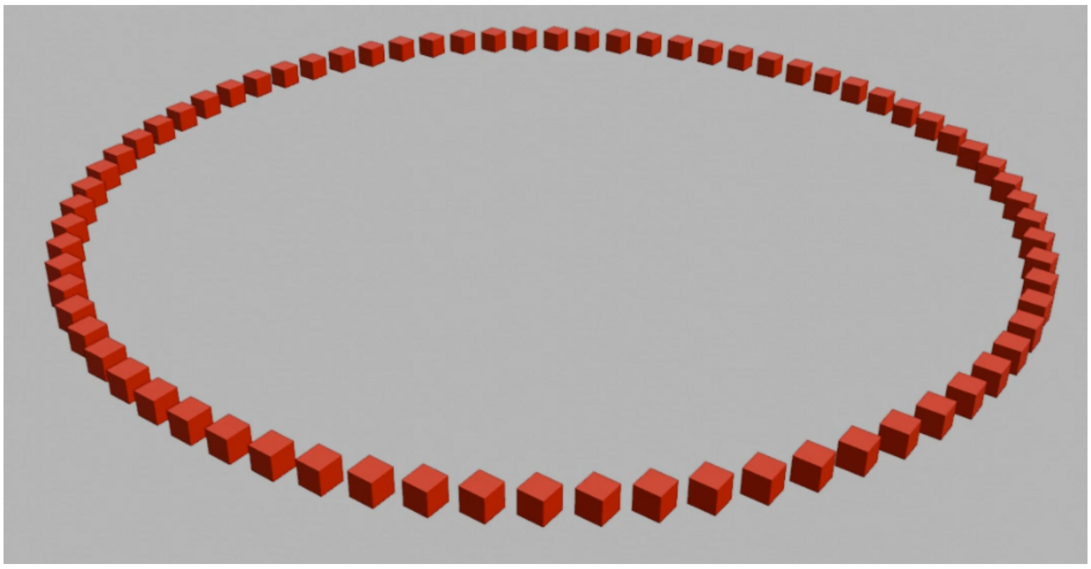


Figure 1.3: Output example

Workflow Architecture

There are four key step to our workflow, namely:

1. Writing the codes we will be tracing. This will be further discussed in chapter 3
2. Running them on our testbed and generating their trace
3. Extracting the trace file and staging its data for a later use with Blender (nonetheless the output is compatible with any other tool) through a dedicated Python script
4. Running Blender's Python script to generate the scene and the final video.

2.1 Trace Generation

To trace our programs, we will be using `Score-P`[15], a powerful performance profiling and tracing tool designed for high-performance computing (HPC) applications. With it, we collect detailed execution data from parallel applications and produce performance traces in the OTF2 file format. Since Vampir reads that same file format, we can easily compare both visualizations generated from the same file.

Alternatively, we can also use `DisCostiC-Sim`[4], a cross-architecture simulation framework designed to predict performance of MPI programs on HPC systems. It simulates computation and communication across cores and nodes, without executing the target program on real hardware, and is able to output to an OTF2 file as well, and was in the earlier development phases used to explore OTF2 files and write our extraction script.

Setting up Score-P

We use Score-P's compile-time tracing option, rather than the runtime option. This requires some key modifications to the standard workflow, mentionned in the Score-P Cheat Sheet [14], specifically:

- Loading the dedicated Score-P module and its dependencies with `module load scorep`
- Exporting `SCOREP_ENABLE_TRACING=true`, to actually enable the tracing, since by default, Score-P only does profiling.

- Exporting `SCOREP_TOTAL_MEMORY=3GB`, to increase the memory per process dedicated to Score-P's tracing, which is close to the maximum. This is to avoid as much as possible costly flushes that have an impact on the program's performance and change its behavior, although this is orders of magnitude higher than what's usually necessary.
- Exporting `SCOREP_EXPERIMENT_DIRECTORY=scorep_run_trace_folder` to optionally set the Score-P output folder, but it can also be overlooked since Score-P creates a default folder.
- Optionally exporting `SCOREP_FILTERING_FILE=scorep_filter_file` so that Score-P ignores specific regions listed in a filter file. This needs to be also set in the compilation command for instrument filtering, failing to do so would provoke a runtime filtering. More information is available in the Score-P documentation [15].
- Compiling with the `scorep` command as prefix for the normal compiling command, or optionally, `scorep --instrument-filter=./scorep_filter_file` when using a filter, an example of which is in section A.3

Example Compiling Command

Normal compiling command:

```
mpiicc test.c -o binary
```

Corresponding Score-P command:

```
scorep --instrument-filter=./scorep_filter_file mpiicc test.c -o binary
```

Dedicated scripts

For more practicality, two standardized scripts were created.

Batch Job Script

Firstly, a batch job script available in section A.1 was created, handling all the details, namely:

- All Sbatch options for node, socket and CPU requirements, frequency fixing, CPU binding
- Loading the appropriate modules
- The compilation through the dedicated Makefile
- Setting up the environment for Score-P and for the generated trace's extraction of data, later used in Blender
- The actual extraction of that data through our dedicated script
- Saving the used code, binary, job script, input file, Makefile, standard output and standard error as well as full environment set-up for later reproduction or debugging

Makefile

Secondly, a standardized Makefile available in section A.2, handling:

- Using the `scorep` command
- Optionally using a filter file, as described in section A.3
- Providing standard flags for debugging, OpenMP, libraries, warnings, speed optimizations, macros
- Deleting intermediary files and the binary to clean the code directory

2.2 Trace Extraction

MPI trace information is parsed and extracted using a Python script converting it to a text file consumable by Blender. It takes as input an OTF2 format trace file.

2.2.1 The OTF2 format

The OTF2 (Open Trace Format 2) is a widely used, scalable, and efficient trace file format designed for storing all execution data from parallel applications using MPI and OpenMP, to which it binds itself, to track the execution of each key step such as entering/exiting a region, which correspond to functions in the code, sending/receiving messages, I/O operations, OpenMP thread creation, etc. All these specific actions in the execution are recorded within so-called events that represent them, while associating all the corresponding metrics, in particular the timestamps for such execution.

For this, we will be using the dedicated OTF2 Python Interface and its documentation [12] [10] [11] to read and prepare all the necessary data from the generated trace file to be later read and handled by Blender.

We'll focus on two specific event type:

- `Enter` and `Leave` events track the entry and exit points of code regions, and provide their relative timestamps.
- `MpiSend`/`MpiRecv`, and `MpiIsend`/`MpiIrecv` events track the transfer of MPI messages, and their timestamps, for respectively synchronous and asynchronous sends/receives between processes.

In the context of asynchronous messages, the OTF2 format implements further details for the different stages regarding a message being passed from a process to another:

- An `MpiIsend` event indicates that a non-blocking MPI send operation was initiated.
- An `MpiIsendComplete` event indicates the completion of a non-blocking MPI send operation.
- An `MpiIrecvRequest` event indicates that a non-blocking MPI receive operation was initiated.
- An `MpiIrecv` event indicates the completion of a non-blocking MPI receive operation completed.

2.2.2 Extraction Script Usage

There are actually two main uses for our script. The standard and default use was already introduced: it is the extraction of the relevant trace data and timestamps later used in Blender. But in order to read and investigate an OTF2 trace file, it is also possible to output all binary content to a human-readable format. This is especially useful for debugging when implementing new features or updating the execution behavior. Therefore, there are two possible prefixes for each mode, defining the behavior of the script.

`--mode (str)` **Description:** Determines which output files will be generated. The modes are:

- `read_full`: Reads the full trace data for all ranks.
- `read_rank`: Reads rank-specific trace data. This needs the `--selected_rank` option to be defined.
- `read_rank_formatted`: Reads rank-specific trace data and outputs to a csv file while outputting timestamps minus the initial timestamp, and ignoring much of the data for readability. This needs the `--selected_rank` option to be defined.
- `read_formatted`: Reads the formatted trace data for all ranks and outputs to a csv file while outputting timestamps minus the initial timestamp, and ignoring much of the data for readability.
- `extract`: Generates the `Ranks.txt` and `CompStopAndStart.txt` files, used in Blender
- `extract_limited`: Generates `Ranks.txt` and `CompStopAndStart-limit.txt`, which limits to a certain number of event or a certain timestamp, given by the user to ignore every subsequent trace event, therefore limiting the output file. This needs the `--cycle_maxcount` or the `--loop_maxcount` to be defined.
- `extract_full`: Generates `Ranks.txt`, `CompStart.txt`, `CompStop.txt`, and `CompStopAndStart.txt`. This can be used for implementations requiring separate files.
- `extract_discosim`: This options allows using traces generated with `DisCostiC-Sim`[4] [3][2].

Default: `"extract"`

`--root_folder (str)` **Description:** Defines the relative path to the root folder containing the Score-P trace data. The output files will be saved in this folder, separate from the trace data.
Default: `"./"`
Example: `"raw_data/myprogram/"`

`--scorep_folder (str)` **Description:** Defines the folder used by Score-P to output trace files. This folder contains the raw trace data generated by Score-P.
Default: `"scorep_traces/"`

`--selected_rank (str)` **Description:** Specifies the rank for which the trace data should be extracted. If not provided, no rank-specific data is extracted.
Default: `None`

`--cycles_per_frame (int)` **Description:** Specifies the number of cycles per frame in the Blender video generation. This value affects the precision and granularity of events shown in the video and is used to calculate the time-flow difference between the execution and the video.
Default: `100000`

`--cycle_maxcount (int)` **Description:** The upper limit for the cycle timestamp in the timestamp output files. If the value exceeds this threshold, the cycle timestamp will be truncated.
Default: `math.inf` (Infinity)

`--loop_maxcount (int)` **Description:** Defines the maximum number of lines in the timestamp output files. This limits the number of frames/steps that are included in the output.
Default: `math.inf` (Infinity)

`--cpu_frequency (int)` **Description:** The CPU frequency in Hz. This is used to convert timestamps into real-time and is used to calculate the time-flow difference between the execution and the video.
Default: `2400000000` (2.4 GHz)

`--framerate (int)` **Description:** Specifies the number of frames per second (FPS) for the Blender video output. This affects the smoothness and speed of the video, and is also used to calculate the time-flow difference between the execution and the video.
Default: `60`

`--fading (int)` **Description:** Defines the number of frames for a color transition (fading) in the Blender video. This is used to control the visual effect of changing colors in the animation.
Default: `1`

`--threshold (int)` **Description:** The threshold number of frames for a time jump. If there is inactivity (no events) for more than this number of frames, the script will compress time in the video output to skip the idle period.
Default: `10000000`

Example Command

```
python trace_converter.py --root_folder=$TRACE_CONVERTER_ROOT_FOLDER \
--framerate=60
```

2.3 Blender Scripting

Blender's Python API is used to:

- Import data from the timestamp files
- Generate all the 3D objects necessary for our representation, according to the user's selected geometry
- Color these objects depending on their MPI status along a series of frames that will compose the output video
- Use Blender's capabilities for creating comprehensive and agreable textures and lighting, as well as camera angles and movements within our 3D environment scene.
- Render the output to a video

Key note: since there are an enormous amount of possible communication arrangements, and an appropriate and comprehensible 3D arrangement would be either hard to guess or even non-existent for the oddest ones, our second script executed within Blender is not designed to guess the correct geometry from the communication patterns exhibited in the trace files. Instead, it focuses on generating some classic and usual domain shapes, such as 2D and 3D grids. Outside of those, the user is supposed to give the appropriate 3D disposition of the ranks.

2.4 Video Generation

After running the Blender script, a simple click on `View > Viewport Render Animation` automatically generates the video. In particular, this rendering method is computationally much less costly, as the aspirations of this tool do not require any costly ray tracing capabilities for beautiful light reflection effects or fuzzy transparency.

Implementation

3.1 Used Code

To safely assert the usability of our tool, we made sure to use trace data collected from real-world runs of applications that are actually used in scientific simulations. For this, we chose to solve a Poisson equation using the iterative Jacobi algorithm. All used codes are available in the A.4, A.6, and A.5 sections of the appendix.

3.2 The Jacobi method

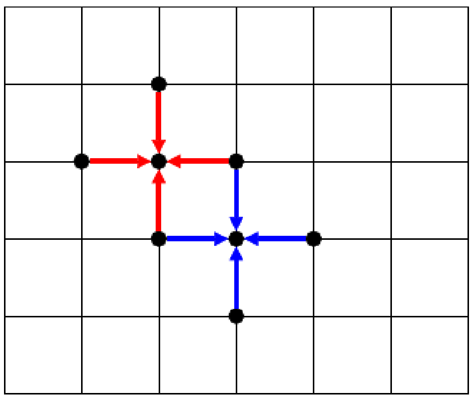


Figure 3.1: Graphical representation of the Jacobi rank communication

The Jacobi method is an iterative algorithm used to solve systems of linear equations of the form $A\mathbf{x} = \mathbf{b}$, where A is a square matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the right-hand side vector. The Jacobi method assumes that the matrix A can be decomposed into its diagonal component D and the remainder $R = A - D$, such that the iteration formula is given by:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}).$$

For each component i of the solution vector $\mathbf{x}$, the update rule can be written as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right),$$

where a_{ij} are the elements of matrix A. The method proceeds iteratively, updating each component of $\mathbf{x}$ based on the previous iteration values. The Jacobi method is simple and parallelizable.

The key component that is necessary for our tool, and that makes the Jacobi our chosen algorithm, is its necessity to send and receive data from its direct neighbors in each iteration of the solver, which we will use, through an extra workload on a certain rank, to create an idle wave.

3.3 Idle Waves

To assess the effectiveness of our various visualization proposal, and to demonstrate their usefulness, we must select a well-fitting case for which they would prove practical. Generating idle waves seems ideal for this task.

Idle waves[7][1][5][6] are a performance phenomenon in parallel computing where idle time, typically caused by synchronization delays or imbalances, propagates across a set of processes in a wave-like fashion. This effect is most commonly observed in tightly coupled MPI applications, where processes frequently wait on one another to exchange data. If one process becomes delayed, due to a hardware interruption, system noise, or uneven computation load, it may arrive late at a communication point, causing its neighbors to wait. These delays can then ripple outward through the system, forming a pattern of staggered idle periods.

In traditional tools such as Vampir, idle waves often appear as stair-like diagonal bands, indicating how the idle state moves across process ranks over time. This effect, while very visual, can be hard to inspect on Vampir, especially in the context of a high number of ranks, or higher-dimension communication, even if it is only 2D. This is a very visual

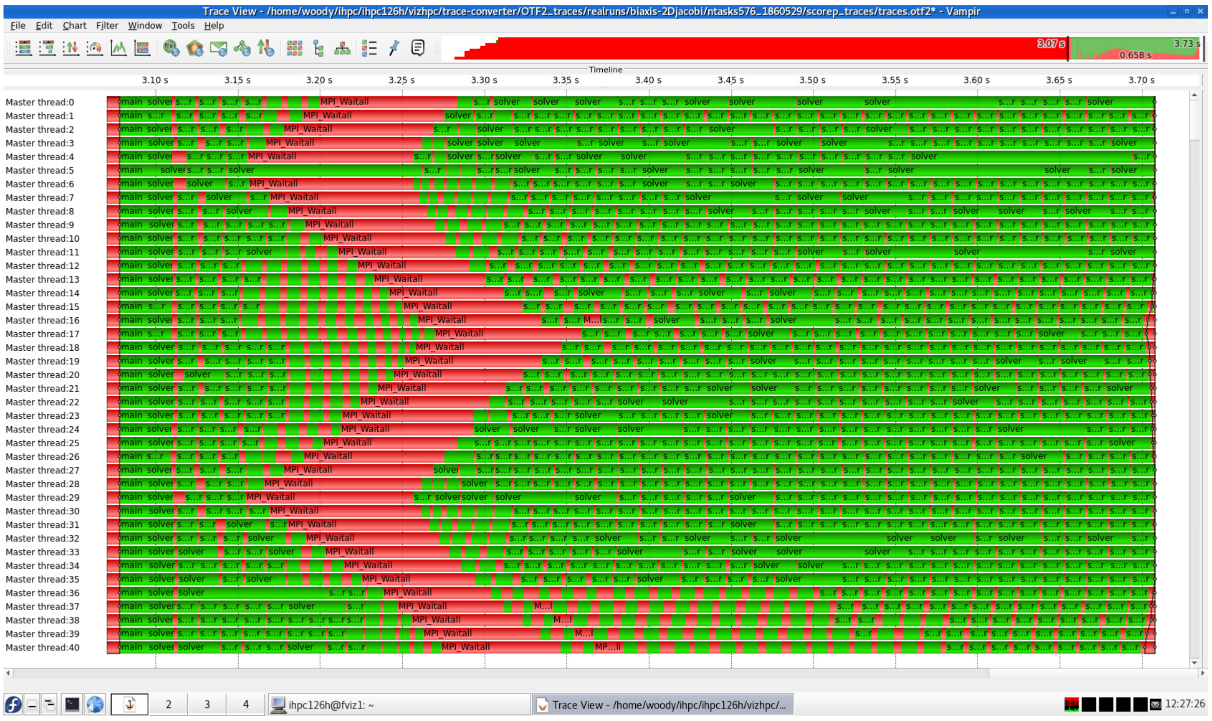


Figure 3.2: Example of idle wave in Vampir

effect, and it is typically what could be significantly more straightforward to view with our visualization, especially in the context of higher dimensional communication since it is particularly difficult to comprehend with existing trace visualization software and illustrate the usefulness of the tool we're developping. Hence, it is the ideal phenomenon to display with it. Also, since the presence of idle waves is particularly concerning in large-scale HPC applications, as they can heavily degrade parallel efficiency, it gives a interesting use case for our tool being used in actual HPC research to understand and mitigate idle wave propagation.

In every instance, we simulate this phenomenon by insuring our Jacobi codes are memory-bound by assigning 20MB of data for each process, ensuring us that this is significantly over their cache capacity and that they will need to fetch it from the RAM, and then injecting an extra workload on rank 5, from which the ripple effect will originate. This ripple effect is what we hope to show in our visualization that would display it in a way that would be both elegant and instantaneously understood, with details being straightforwardly shown while they are hard to grasp in Vampir.

Testbed

To run our programs and generate the relative trace files, we used the Fritz cluster[9]. This testbed environment provides the conditions for controlled and reproducible experiments that can accurately be recreated.

Then, to run Blender 4.3.2, we used either a standard office laptop, or the , slightly more powerful `fviz1` remote visualization nodes available on Fritz.

4.1 Hardware Setup

The testbed consists of the following hardware components:

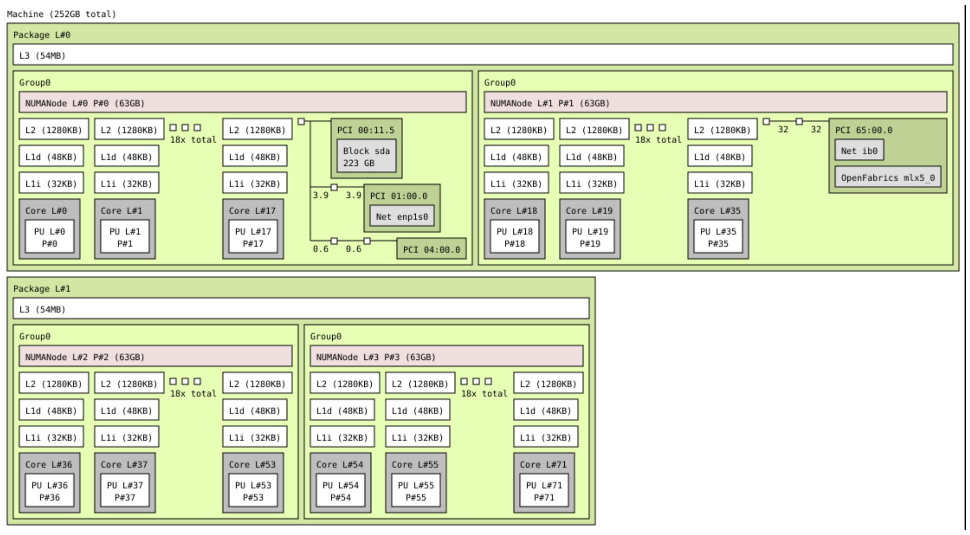


Figure 4.1: Topology of Fritz

- **Nodes:** Each node is composed of 2 x Intel Xeon Platinum 8360Y (”Ice Lake”), each containing 36 cores @ 2.4 GHz. Simultaneous Multithreading (SMT) is deactivated, therefore a single thread will be run per each core. The 36 cores can be further divided into 2 CC-Numa domains of 18 cores each. On the total of 992 nodes, we can allocate up to 64 nodes at a time for a job.
- **Memory:** Each node processor has 54MB of L3 cache, for a total of 108MB for the two processors. Furthermore, each core has a 1MB L2 cache, for a total of 36 MB per processor. Each node is equipped with 256GB of DDR4 RAM @ 3200 MHz.

- **Network:** An HDR100 Infiniband with up to 100 GBit/s bandwidth.
- **Clients:** Five desktop machines with Intel Core i7 processors and 16 GB RAM, simulating user workloads.

4.2 Software Environment

The software stack installed on the testbed includes:

- **Operating System:** AlmaLinux 8.
- **Manager:** Slurm Workload Manager.
- **Modules:** We compiled our codes using `intel/2021.4.0` with C11 standard. Our MPI vendor is `intelmpi/2021.7.0`. We used version 8.1 of Score-P for profiling and generating our trace files, under the name:
`scorep/8.1-intel-2021.4.0-intelmpi-2021.7.0-papi`.
- **Run program:** We use self-developped codes implementing or mimicking the aforementionned Jacobi. There is a 2D, a 3D and a spherical versions, all in C.

4.3 Test Scenarios

We tested several task binding/pinning schemes to find out if the effect of such pinning would be displayed in a noticeable way in our visualization, and especially how straightforwardly and comprehensively this would show up.

4.4 Fviz1 Nodes

`Fviz1` was used for remote visualization with VirtualGL, in order to run Blender, generate and display its scene. It consists in an Nvidia A16 GPU, partitioned into 4 virtual GPUs, one of which we attach our VNC client to. It offers 16GB of RAM, and runs on Xfce Linux.

4.5 Personal Laptop

A simple office Dell laptop was used

- **CPU:** 11th Generation Intel® Core™ i5-1135G7 (8 MB Cache, 4 Core, 8 Threads, 2.40 GHz to 4.20 GHz, 17.5 W).
- **Memory:** 16 GB, DDR4, 3200 MHz, dual-channel, integrated.
- **Storage:** M.2 2280, 1 TB, Gen 3 PCle x4 NVMe, Class 40 SSD.
- **Graphics:** Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics].
- **OS:** Ubuntu 24.04.2 LTS.

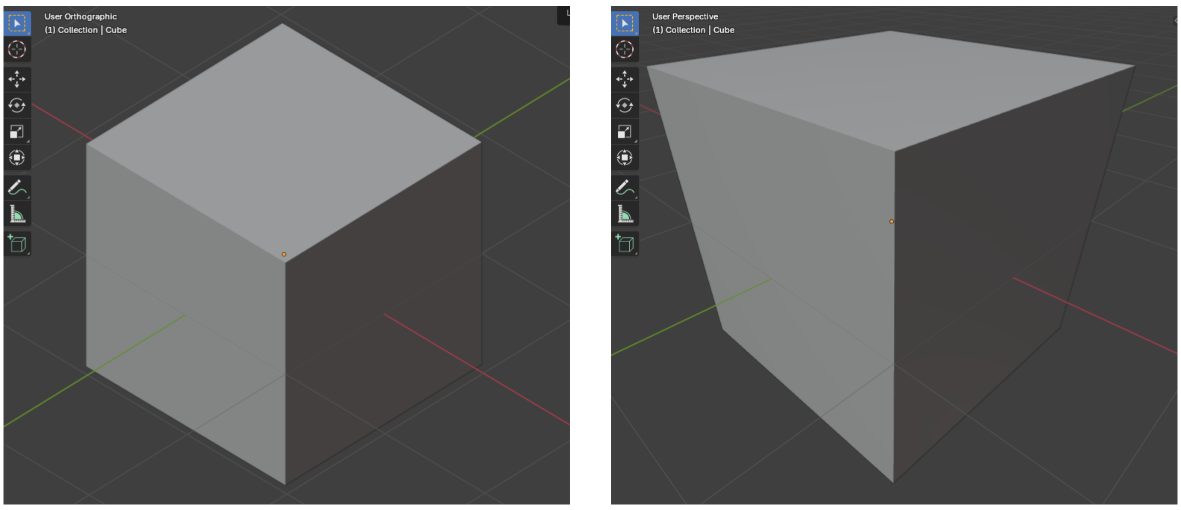
3D Visualization Implementations

Some choices made the 3D visualizations more intuitive for understanding complex messaging behavior compared to traditional tools.

5.1 Perspective vs Orthographic Projection

Blender offers two ways to display any object, the user has to select between a perspective projection or an orthographic projection, otherwise known as isometric projection, which are in 3D graphics and technical drawing, the two different ways of representing 3D objects on a 2D surface such as a computer screen.

- **Perspective Projection** mimics how we see in real life. Objects appear smaller as they get farther away, creating a sense of depth. It's commonly used in games, animation, and realistic renderings because it feels natural to the eye.
- **Orthographic Projection**, on the other hand, shows objects without perspective distortion. Parallel lines stay parallel, and size remains constant regardless of distance. This projection is used in blueprints, CAD, and technical drawings, where accuracy and scale are more important than visual realism.



(a) Orthographic projection (b) Perspective projection

Figure 5.1: Base Blender cube

5.2 Different Geometries for Different Domains

5.2.1 Line

This is the simplest case and does not present any additional value compared to the Vampir view, outside of a more practical view of a high number of processes, a full node is shown in Figures 5.2 and ??.

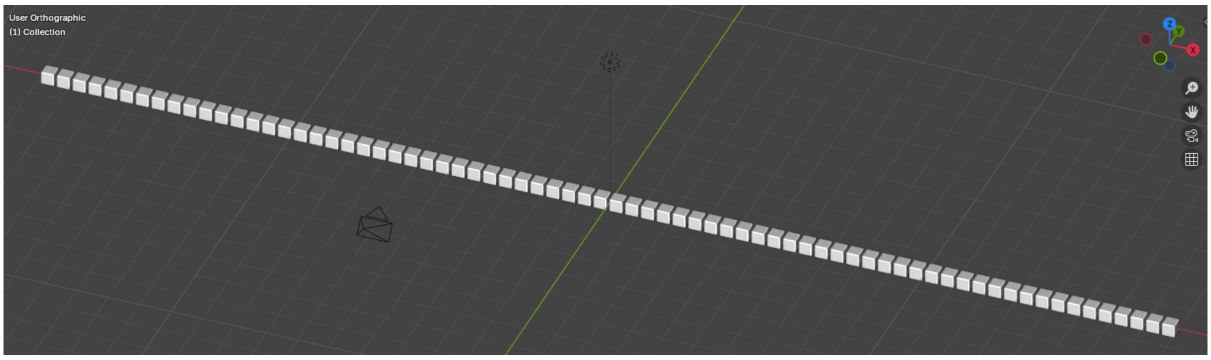


Figure 5.2: Line of 72 ranks in orthographic projection

5.2.2 Circle

We can extend the line to a circle for the case of a periodic communication, where the first and last ranks are directly communicating.Again, a full node is shown in Figures 5.3 and 5.4.

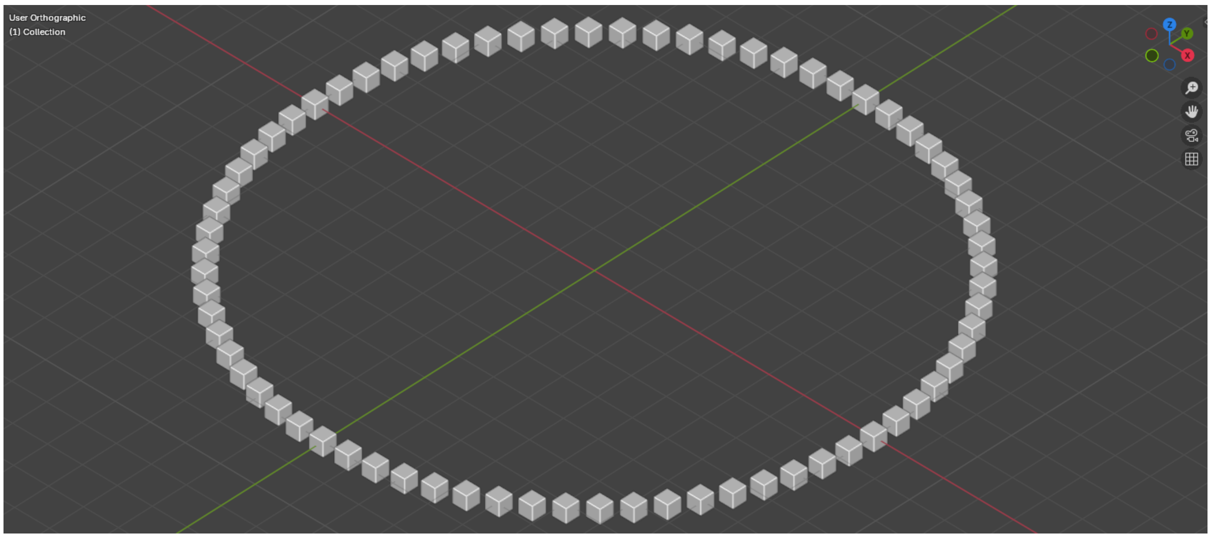


Figure 5.3: Circle of 72 ranks in orthographic projection

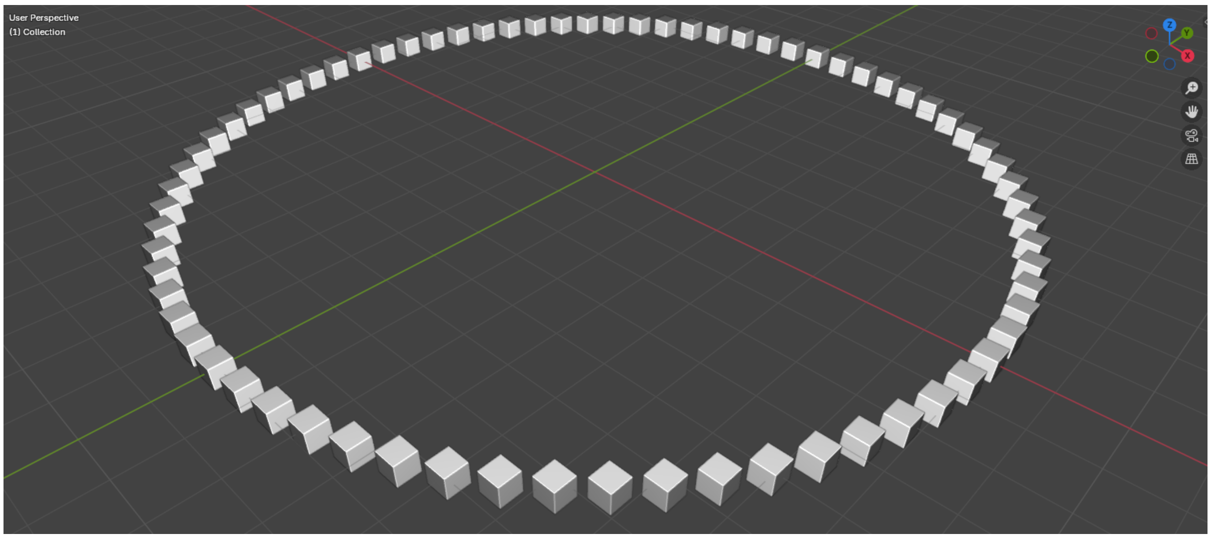


Figure 5.4: Circle of 72 ranks in perspective projection

5.2.3 Grid

The case of the grid is the first and most frequent case of 2D communication, where ranks are connected both with their left and right neighbor as well as with their up and down neighbors. This time, Figures 5.5 and 5.6, are showing a grid of 24x24 ranks, for a total of 576 ranks, composing 8 nodes.

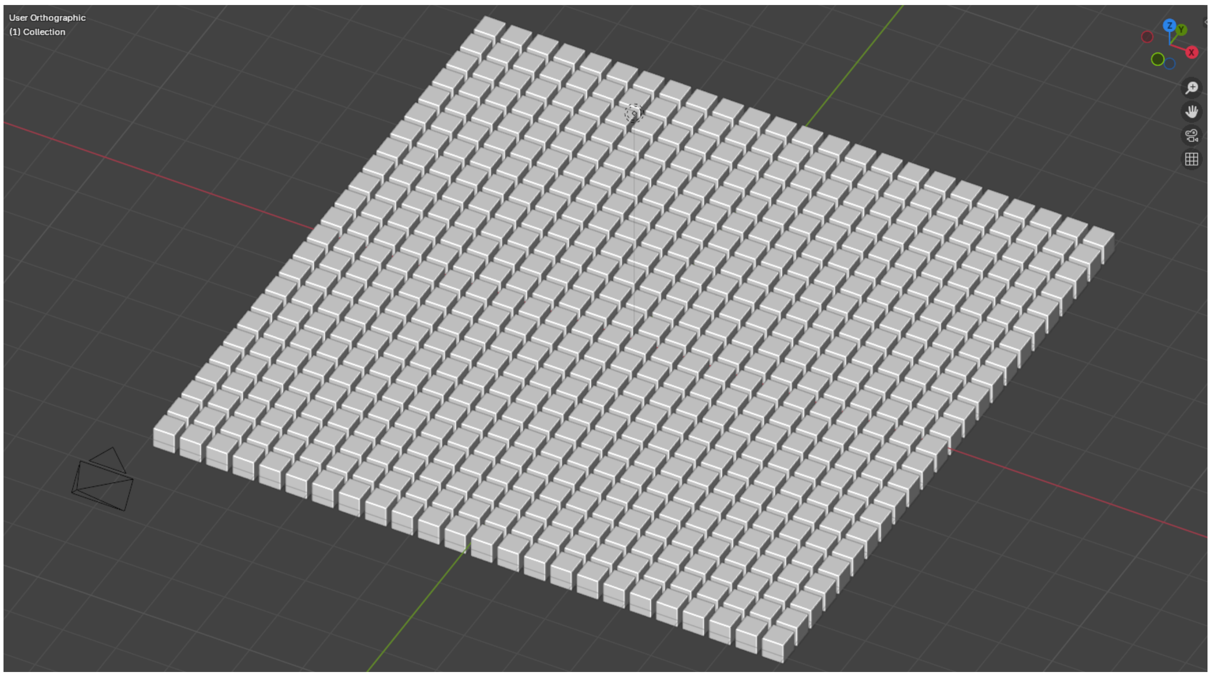


Figure 5.5: Grid of 24x24 ranks in orthographic projection

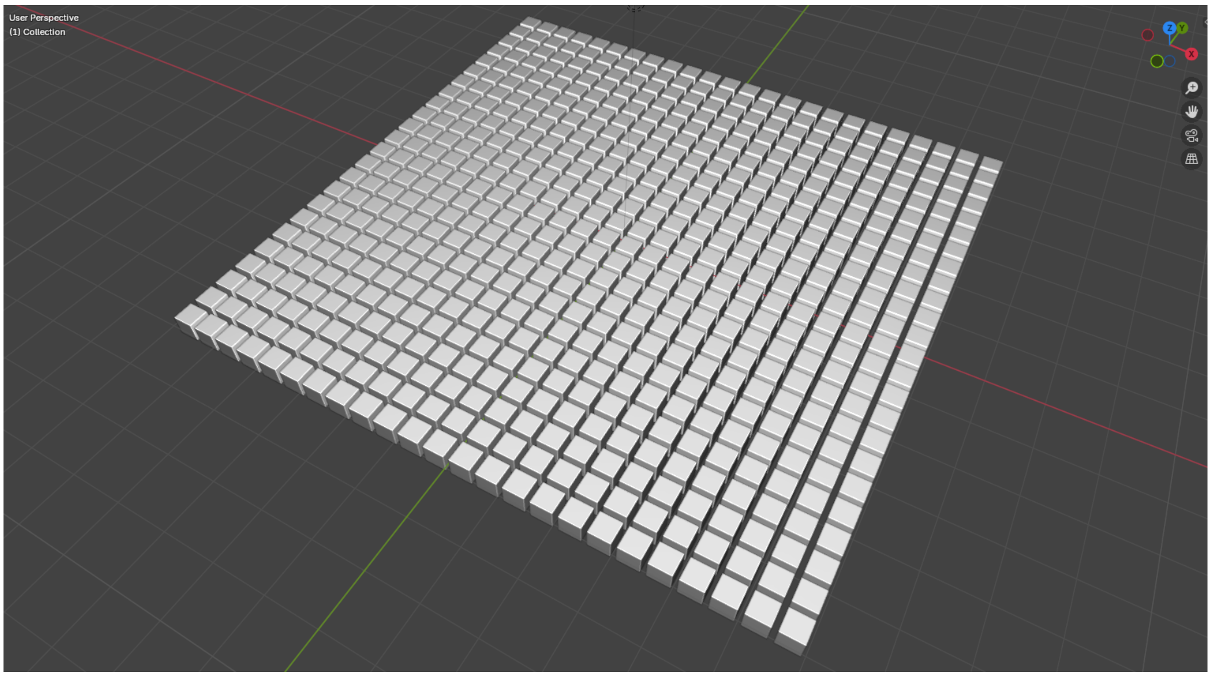
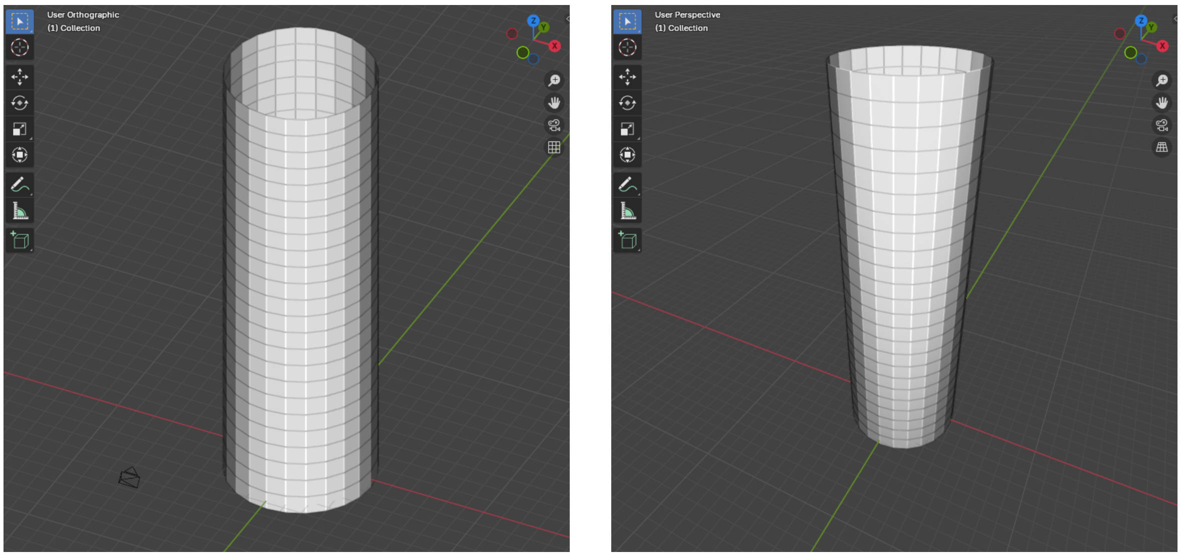


Figure 5.6: Grid of 24x24 ranks in perspective projection

5.2.4 Tube

The tube, or hollow cylinder / cylindrical shell, is the same as the grid that has a periodic communication in a single direction. In terms of clarity, it rarely offers an advantage compared to the standard grid, since we can usually intuitively grasp the communication happening on the edges. However, it may be useful for more complicated communication patterns occurring on the boundaries. Figure 5.7 is again showing a grid of 24x24 ranks, for a total of 576 ranks, composing 8 nodes.

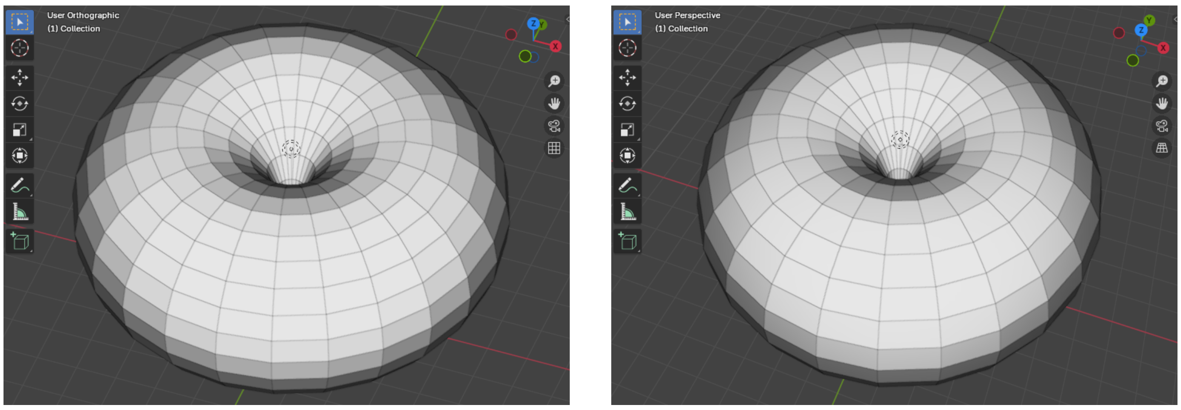


(a) Orthographic projection (b) Perspective projection

Figure 5.7: Tube of 24x24 ranks

5.2.5 Torus

The torus extends further the concept of the tube by adapting to periodic communication in both directions, with ranks at the edges of the grid being connected not only horizontally (as in the tube), but also vertically, forming a seamless wraparound in two dimensions, following communication topologies where the edge effects are eliminated entirely.



(a) Orthographic projection

(b) Perspective projection

Figure 5.8: Torus of 24x24 ranks

However, an obvious problem arises when observing the examples from figures 5.8: the choice for the two radiuses will skew the individual ranks significantly, especially in our case of a square 24x24 rank grid. They can be either very similar or very different, and this will provoke an unavoidable difformation of the ranks' figures composing the global structure. This explains why the torus may not significantly improve clarity for simple communication patterns, and it may become more useful for understanding only some very specific complex or symmetric communication patterns.

Outside of these rare cases, the torus is more adapted to band-like domains, as shown in the next example figure, otherwise, the standard flat display of the grid case is the most straightforward way to better understand communication, including on the boundaries. If more detail is needed, the simple copy of the original grid around itself eight times over, one for every direction, will suffice to make any pattern be both obvious and precisely displayed.

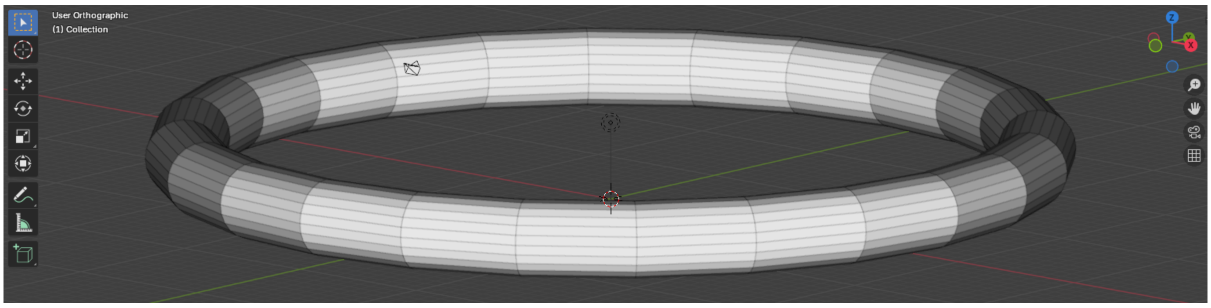


Figure 5.9: Torus of 24x24 ranks in orthographic projection

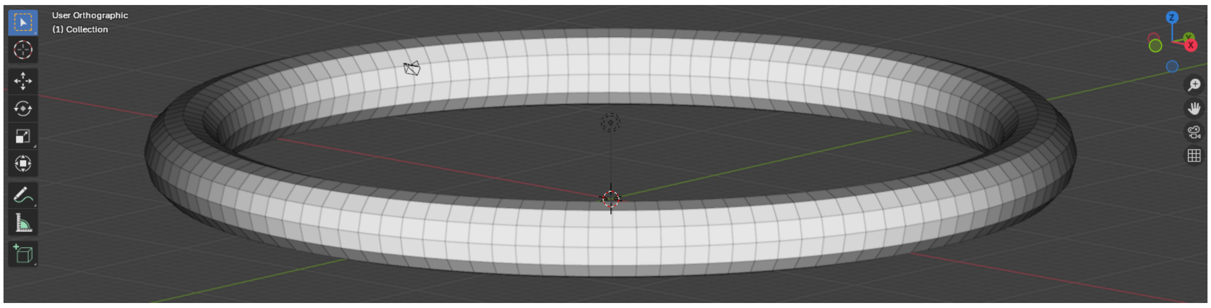


Figure 5.10: Torus of 12x120 ranks in orthographic projection

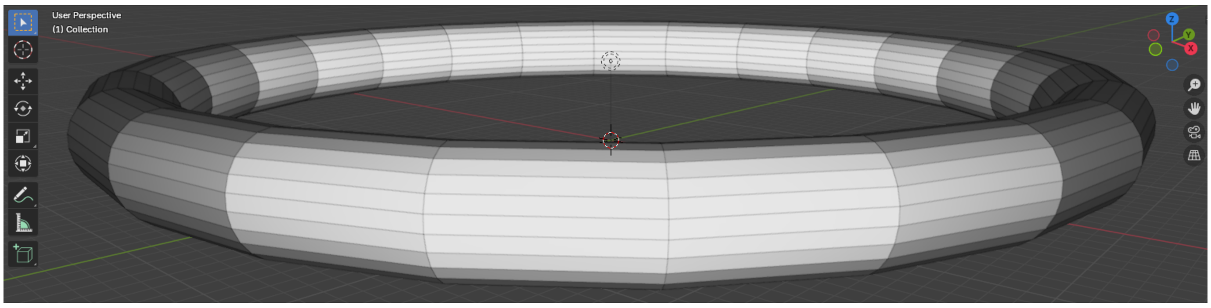


Figure 5.11: Torus of 24x24 ranks in perspective projection

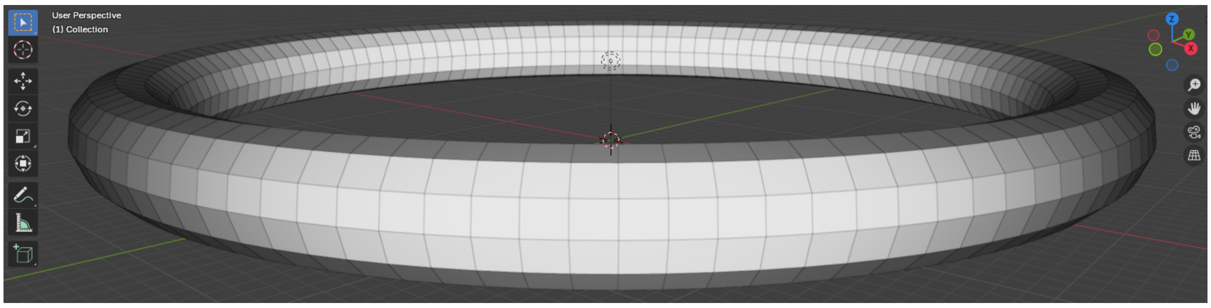
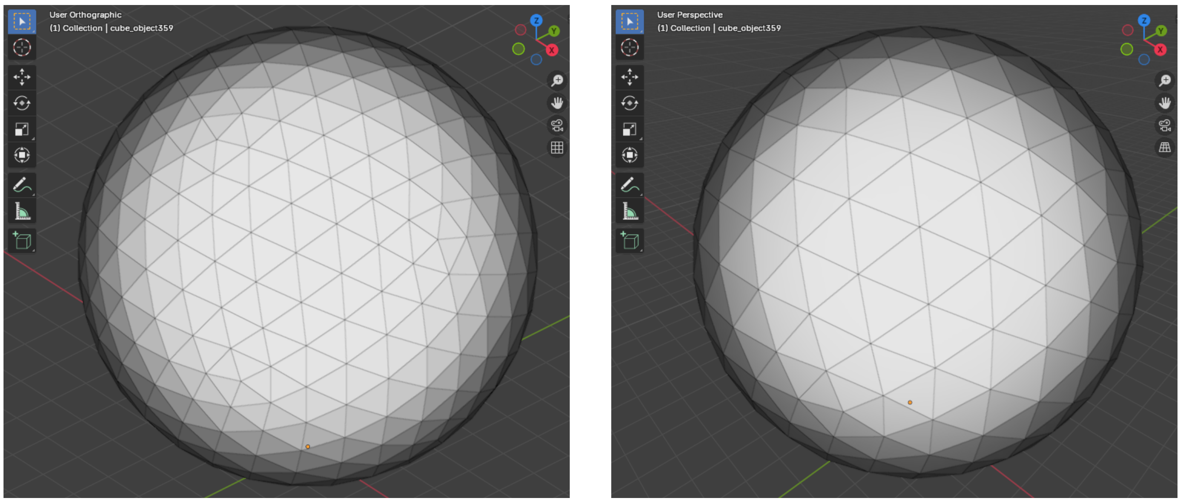


Figure 5.12: Torus of 12x120 ranks in perspective projection

5.2.6 Sphere

The more complex the domain, the more helpful our visualization schemes become. The current case of a spherical domain, while being quite standard, e.g. in Earth simulated phenomenons, will certainly helps us understand better and visualize in a straightforward way the intricate communication schemes between ranks.
To develop a geodesic polyhedron mesh, we started from a standard icosahedron, on which we applied a 6-frequency subdivision on each triangle, and we then projected all vertices onto a sphere. Every triangle is a rank, connected to three other ranks.

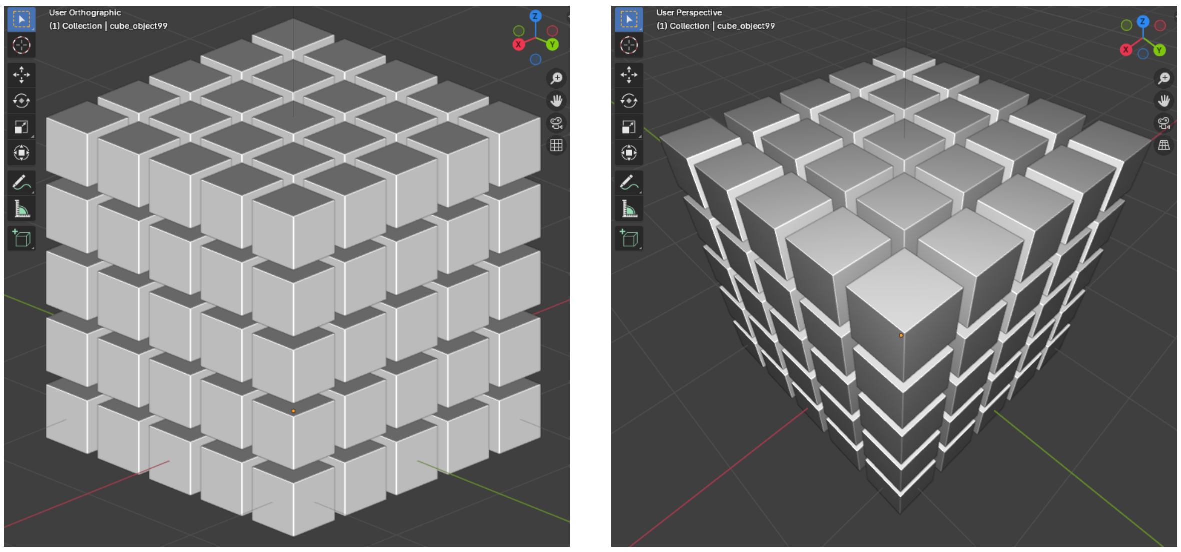

User Orthographic
(1) Collection | cube_object359
User Perspective
(1) Collection | cube_object359

(a) Orthographic projection (b) Perspective projection

Figure 5.13: Geodesic polyhedron of 720 ranks

5.2.7 Cube

Finally, the cube is understandably an crucial addition to our domain list.


User Orthographic
(1) Collection | cube_object99
User Perspective
(1) Collection | cube_object99

(a) Orthographic projection (b) Perspective projection

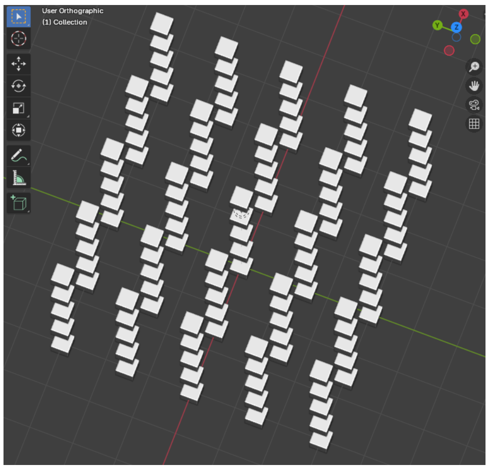
Figure 5.14: Cube of 5x5x5 ranks

Again, some obvious obstacles immediately arise at first glance: we cannot see past the surface. Therefore, we had to come up and test some workaround methods to solve them, which we will discuss in the following section.

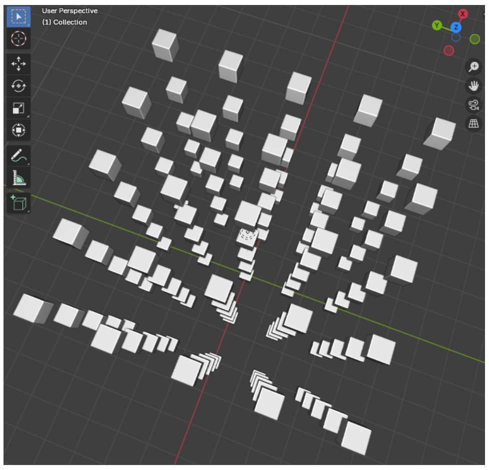
5.3 Display choices

5.3.1 Sparsity

Sparsity, i.e. spacing objects apart in a 3D scene makes it easier to see past the front layer and view what's behind. By avoiding to group objects too close together or in a stacked way, the outer ones doesn't block the view anymore, allowing to see the inner layers, and better understand the overall structure. There is a fine balance in finding the appropriate spacing, as to avoid missing the global structure and the link between the ranks, as well as the view angle. Figure 5.15 shows an update of our preceding cube view.



(a) Orthographic projection



(b) Perspective projection

Figure 5.15: Cube of 5x5x5 ranks with better spacing and view angle

Carefully selecting the angle is a crucial part of such a view, meaning that not all angles are valid in order not to confuse the user.

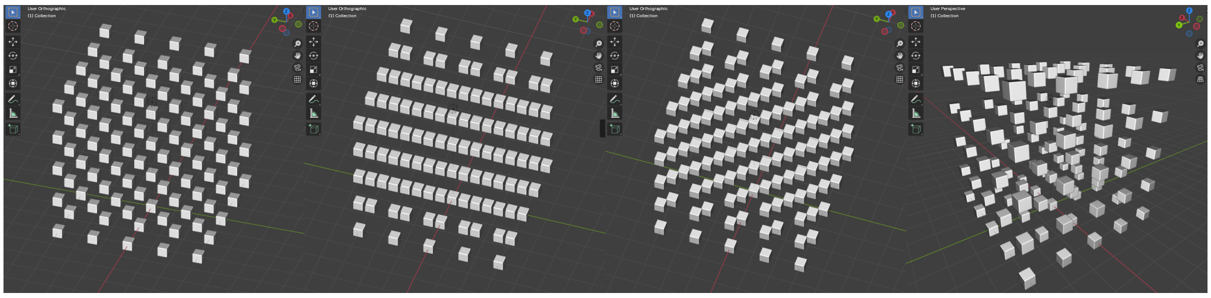


Figure 5.16: Confusing views for the cube

This will lead us to another proposal, in the next section, through camera movements.

5.3.2 Camera movements

In Blender, simple mouse movements can change the camera angle and zoom in and out on the object. These movements can then be coded for the subsequent rendering of the output video. They can also help us mimic 3D on a flat screen, giving us the impression of true 3D, as shown in Figure 5.17.

(a) Orthographic projection
Open in default player

(b) Perspective projection
Open in default player

Figure 5.17: Camera movement around a sparse cube

5.3.3 Displaying MPI and Computation Phases

Following a suiting object creation for our domains, we need to formally define a way to display ranks sitting in or out of MPI regions.

Dual Color Display

The first option we explored was to define two colors, red and blue, respectively for ranks sitting in MPI or not.

Using Transparency

Another idea to better show the difference between processes sitting in MPI and others, and because it further helps, in multi-layer geometries, to see past the first layer, we decided to use transparency to code for any non-MPI event, and to keep red for MPI regions. On later stages of the development and for some geometries, we used a partial transparence also for color-coded MPI events, to be able to see behind their respective Blender objects.

Using a Color Gradient

A key information such as which processes are sitting in MPI for a long time or not, in other words determining laggers and speeders. A tool like Vampir allows the user to see it precisely with a single glance, as well as the sequence in which processes communicate. Using only one color, our tool can only help do that at a high cost, since the user, seeing which processes start communicating, can only know when they stop after the video reaches this point when they finish communicating. Therefore, the user needs to track a rank, or worse, a collection of ranks for that whole time, which can become quickly hard. In particular, a paused video can offer no good insight on the matter.

To help distinguish laggers and speeders, and to know in advance if a process will resume computing soon, we use a red-to-green color gradient: a process that just entered an MPI region will be colored in red, and will gradually swith to green as long as it sits in MPI.

- A **long call** will therefore immediately appear as red and will show little to no change in its color, immediately striking the user as a costly call.
- A **short call** will quickly turn green, or possibly even disappear instantaneously.
- An **intermediary call** will show a gradual turn orange.

This gradient conveys in a simple way to the user an intuition about which communication will take a long time, and helps identify bottlenecks. It essentially gives a premonitory feeling about the upcoming events and the future of the video, without having seen it. It can also even be used as a reminder of the past, displaying which processes have been sitting in MPI for a long time.

With this addition, a very short time playing the video, or potentially even a paused video, gives much insight about how the communication is happening between the ranks, and helps reconstruct the scenario of the execution. This is one of the most important addition to our visualization tool.

Prioritizing Progression over Time

Another potential option to implement the red-to-green gradient change that would have been depending on the amount of time spent in MPI rather than in progression, the user would then know depending on the color for how long in milliseconds the concerned ranks had been in its MPI regions, but this was deemed much less practical or useful, since the user is already at least somewhat aware of that simply with the video playing, and we would lose out the intuitive insight on the future steps of the video. Besides, it also would need us to determine which threshold is appropriate to go from full red to full green.

Blender's Color Gradient Algorithm

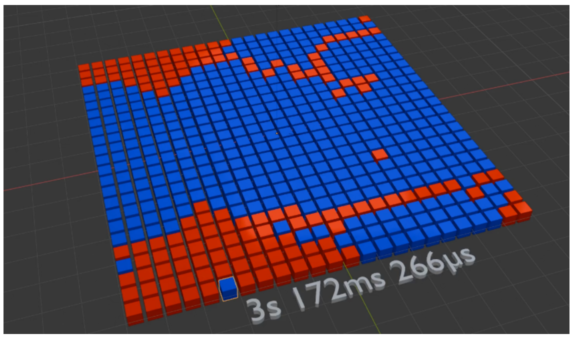
Using Blender's color diffusion method means that any change in the color or transparency of an object isn't defined as happening at a certain timestamp of the trace, but rather between two timestamps. We implemented a `fading` option to be able to select how smooth the object light up and tone down. The standard option is to do it over a single frame, the quickest possible.

Coloring the Extra Workload Rank's Edges?

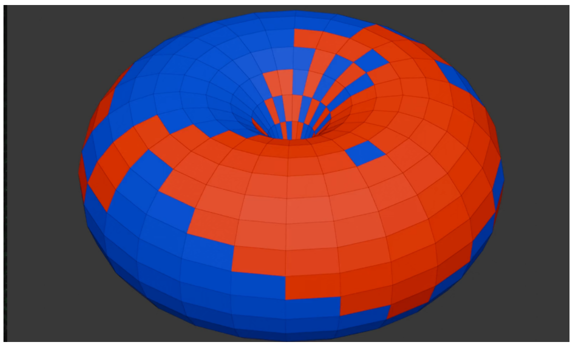
Coloring the edges of our 3D objects can be an interesting addition, they could be used to show in which iteration of a global loop a certain rank is, although this hasn't been completed yet, since we cannot access to a loop counter directly, but instead need to infer it from the region names recorded in the trace. It assumes, possibly wrongly, that all ranks are supposed to call the looping region the same number of times, as well as know that region name in advance.

Results for our different cases

In this section we will be using a 2D-Jacobi code with an idle wave to display the progression of the view over those improvements for the Grid, Torus, Cube and Sphere cases.

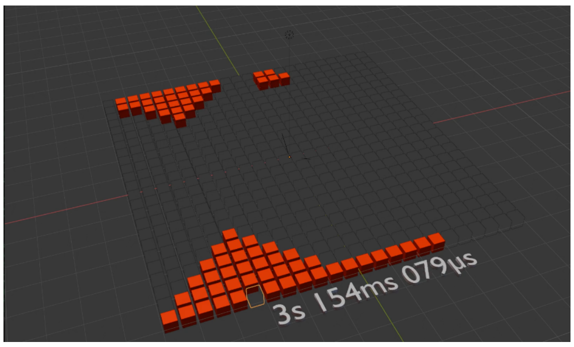


(a) Grid case
Open in YouTube

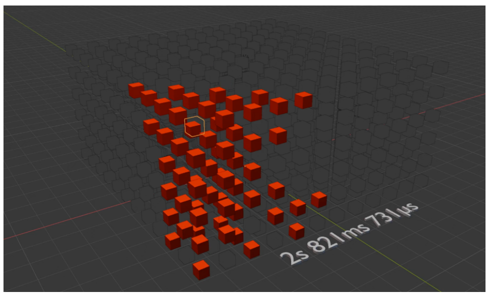


(b) Torus case
Open in YouTube

Figure 5.18: First stage : dual color, red for MPI, blue for computation

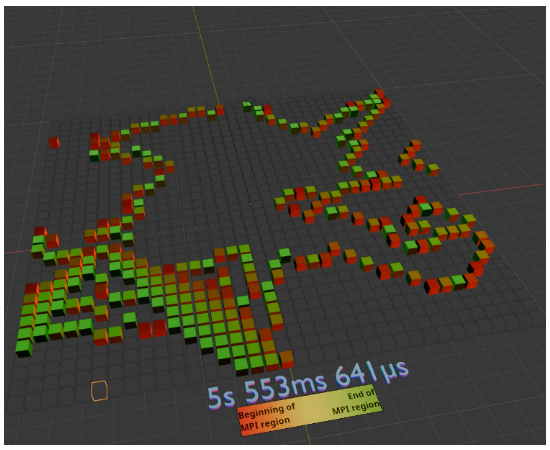


(a) Grid case
Open in YouTube

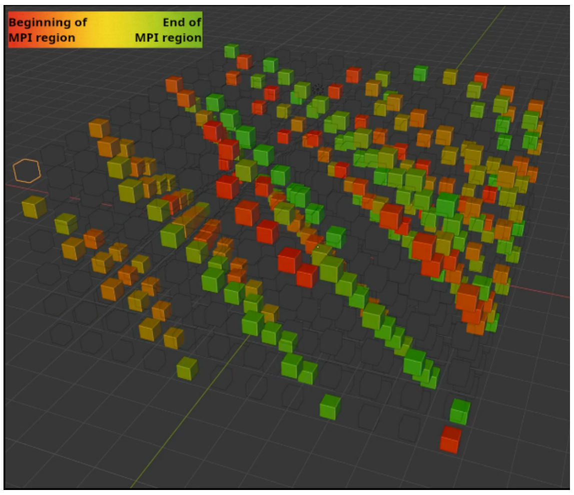


(b) Cube case
Open in YouTube

Figure 5.19: Second stage: Using Transparency for non-MPI regions

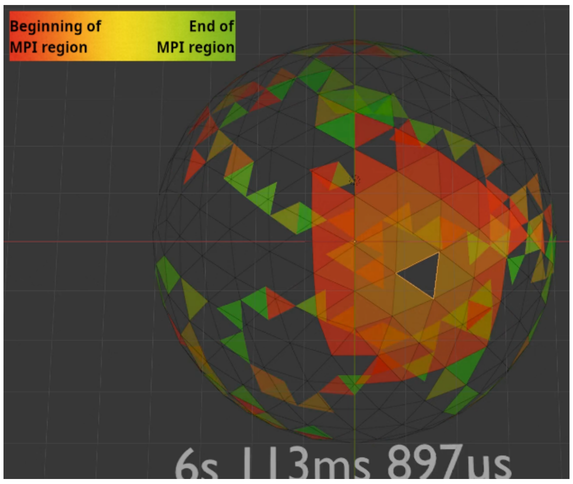


(a) Grid case
Open in YouTube

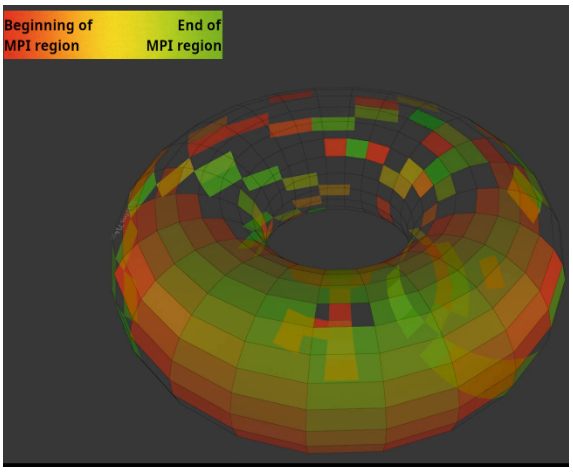


(b) Cube case
Open in YouTube

Figure 5.20: Third stage: Using a Green-to-Red color gradient for MPI regions



(a) Sphere case
Open in YouTube



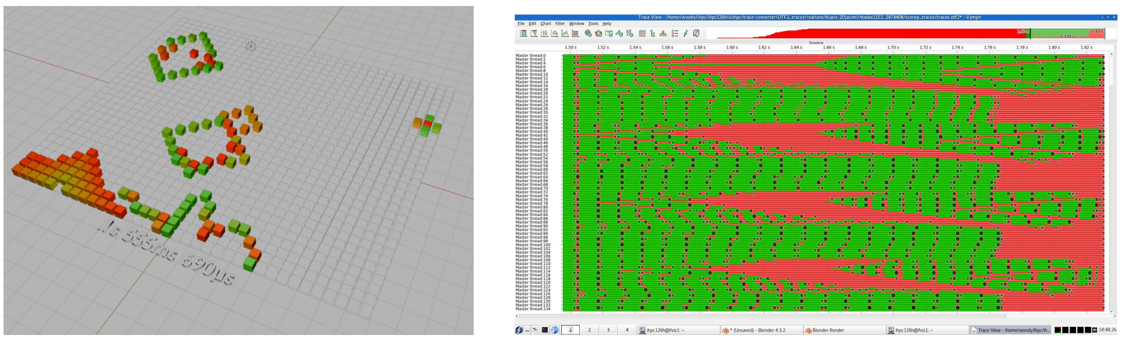
(b) Torus case
Open in YouTube

Figure 5.21: Fourth stage: Using partially tranparent colors to see past them

Comparing Our 3D Visualization with Vampir

Figure 5.2.3 shows a comparison of our visualization with Vampir.

This simple 2D grid run suffices to show the higher dimension problem with Vampir: this somewhat confusing view appears to show multiple rank from which an idle wave originates. It actually is due to the 2D communication, that skips 24 ranks, in other words the size of the line for this grid.



(a) with Blender (b) with Vampir

Figure 5.22: Trace Visualization for a 2D-Grid
Open in YouTube

5.4 Practical additions

5.4.1 Ignoring Uninteresting Trace Events

Initialization Parts

- Any MPI code needs to call `Mpi_Init()`, which is usually uninteresting and significantly long. Since the time length of our visualization is much longer than the actual execution time on the cluster, this leads to an unnecessarily boring and heavy video where nothing happens. This is standardly ignored.

- The user can decide to ignore every event before a specific region, identified by a calling function name. Our Trace Extraction script handles either ignoring every event before the first occurence of such region, or before the last one, as well as ignore that region itself or not.

Relatively Quick Events

Depending on the length of the run corresponding to the trace, and in order to have a digestible output video that isn't impractically long, some events may be too quick to be displayed, since the video discretizes the run over a much fewer number of frames, which is why this number of frame may not be fine-grained enough for two consecutive events or more. Therefore these events happen over the same frame and cannot be displayed. They are lost in translation and do not exist in the output video. The user has the power to increase the framerate, or to make the video span over a longer time, although this may quickly pose scaling problems, or as mentionned before, render the video much less practical or useful.

Time Jumps

Some parts of a trace can be particularly uneventful, and they can be this way for a long time, especially in the output video, where the time is stretched multiple times over. For this reason, we implemented an option to ignore trace parts when no event happen for a long time, making the next event happen much sooner. The threshold to which this feature is triggered is modifiable by the user.

5.4.2 Reproducing the Vampir Visualization in Blender

As discussed in the Introduction chapter, this tool does not claim to replace Vampir-like trace visualizations, only to complement them. They are still very handy, and they are what users are used to. We first proposed to include alongside the video the corresponding Vampir view of the trace, and later decided to go as far as to recreate it in Blender. These trace objects move from left to right, to synchronize the center of the Vampir-like view with the coloring of the corresponding ranks recreated in 3D. Another advantage is the

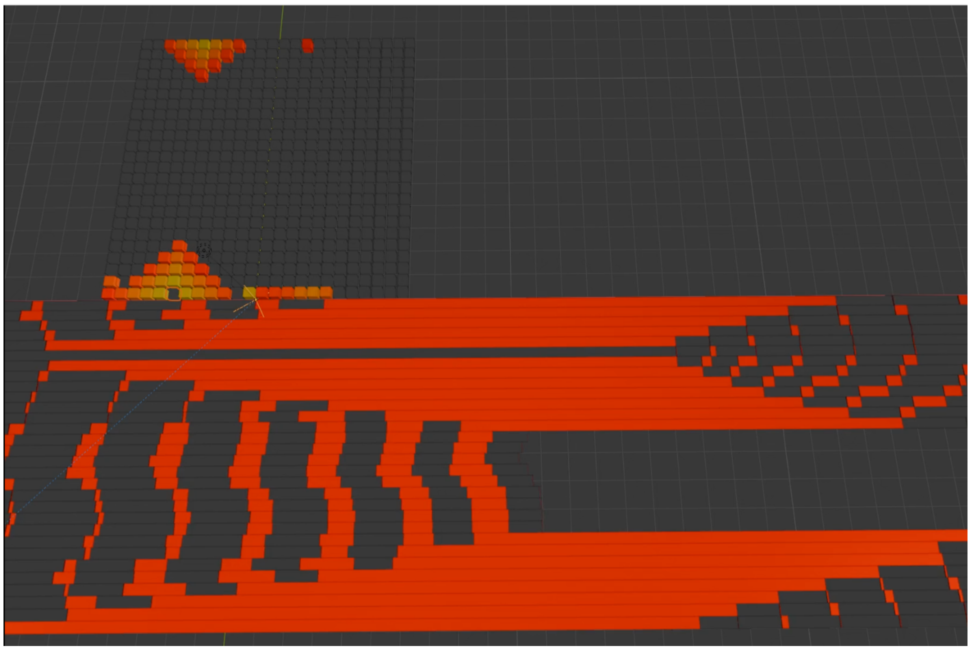


Figure 5.23: Vampir Visualization Reproduced in Blender
Open in YouTube

possibility to rearrange ranks in another order. In particular, for our 2D Jacobi code, we tested a reordering of the ranks from nearest to farthest starting from rank 5, the one where we injected an extra workload and from which we started an idle wave. This changes the Vampir-like visualization to make it look like there is indeed a single idle wave injection, instead of seemingly multiple ones, which is artificially caused by the two-direction communication. However, due to the high number of region switches, creating

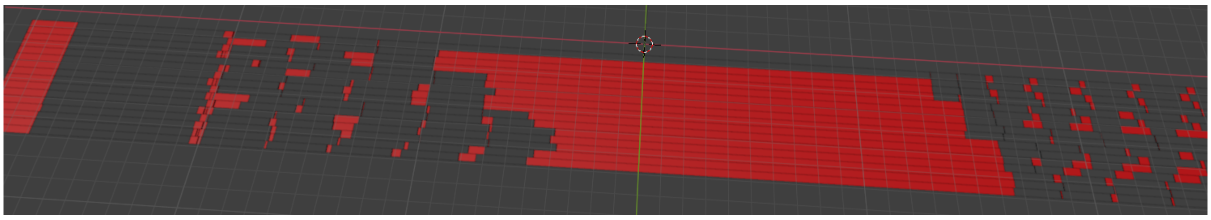


Figure 5.24: Reordering the ranks from nearest to farthest to rank 5

as many corresponding figures in Blender, this is by far the most unstable implementation we developed, and displaying it is always prone to instability issues and crashes. After all, Blender's documentation[8] itself specifies that it is expected to crash over and over again during development.

5.4.3 Communication Arrows

We also wanted to show message being passed. For this, we created arrows. At first, we implemented a solution showing timestamps for `Send/Isend` calls and `Recv/Irecv` calls separately, lighting up respectively the tails, and the heads of the concerned arrows.

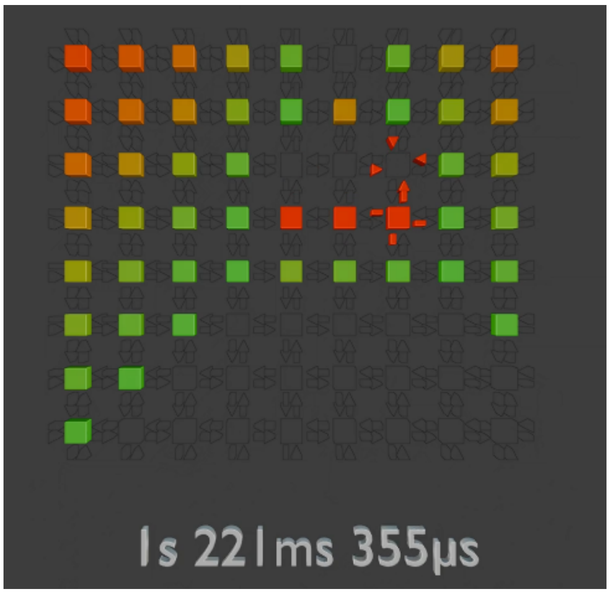


Figure 5.25: Communication Arrows representing MPI messages
Open in YouTube

We later moved to merge the arrows' heads and tails using the same color gradient as for processes, and using timestamps for `Send` calls and `Recv` as beginning and end times.

5.4.4 Showing NUMA domains and displaying the Pinning

Figure 5.26 is an attempt at showing how fast or privileged the communication is between two specific ranks was to space ranks differently with regard to them being or not in the same node, processor or CC-Numa domain. This is a very interesting addition to understand some communication patterns, in particular those being the consequence of a set of ranks pertaining to the same CC-Numa domain and benefitting from faster communication. This interesting view allows to see both the logical topology and the underlying physical cores that are running the tasks, in order to inspect more deeply the intricate details of the trace. Seeing the CC-Numa domains can allow the user to better view the less-costly communication and memory operations, which explains greatly some of the trace's output and the general program behavior. Specifically for this run, we pinned each 3x6 rectangles to a dedicated CC-Numa domain. It soon becomes obvious that their shape, along the X-axis, is responsible for the subsequent X-axis idle wave fronts of the run.

5.4.5 Clock

An obviously useful addition was to include a clock corresponding to the runtime, and displayed in seconds, milliseconds and microseconds for practicality. In order to match Vampir-like tools, it needs to keep as origin the initial start of the trace, which is why it usually doesn't start at 0 seconds.

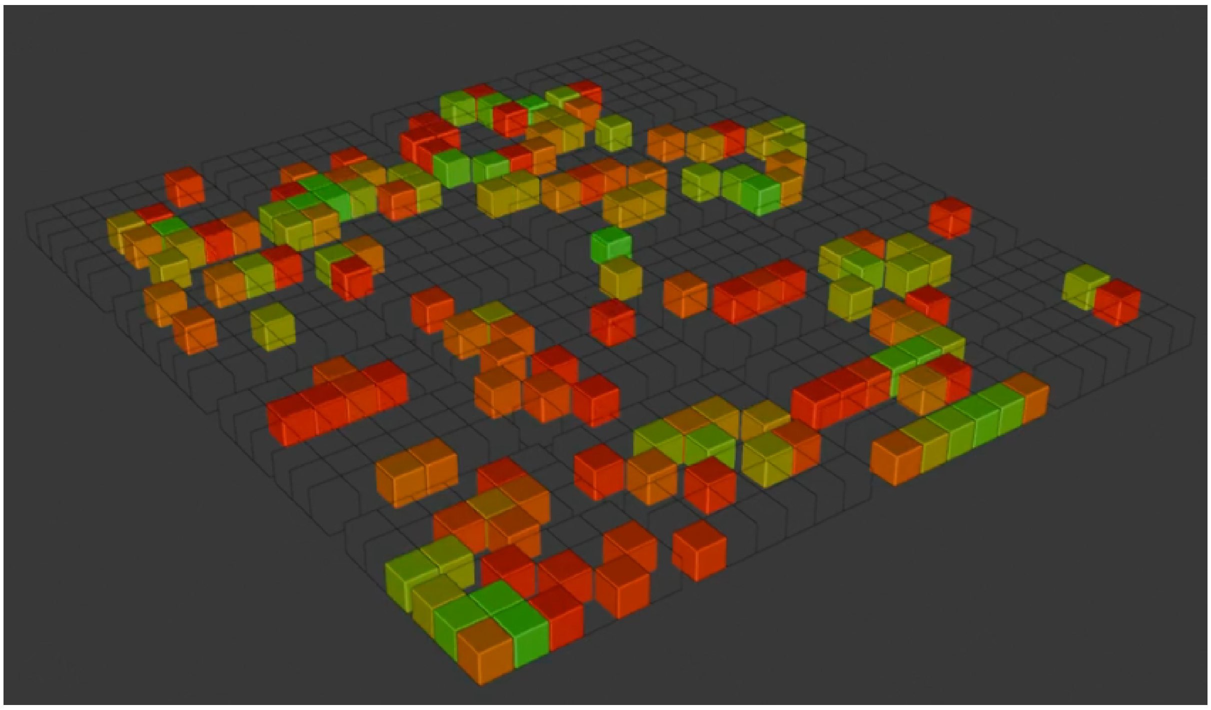
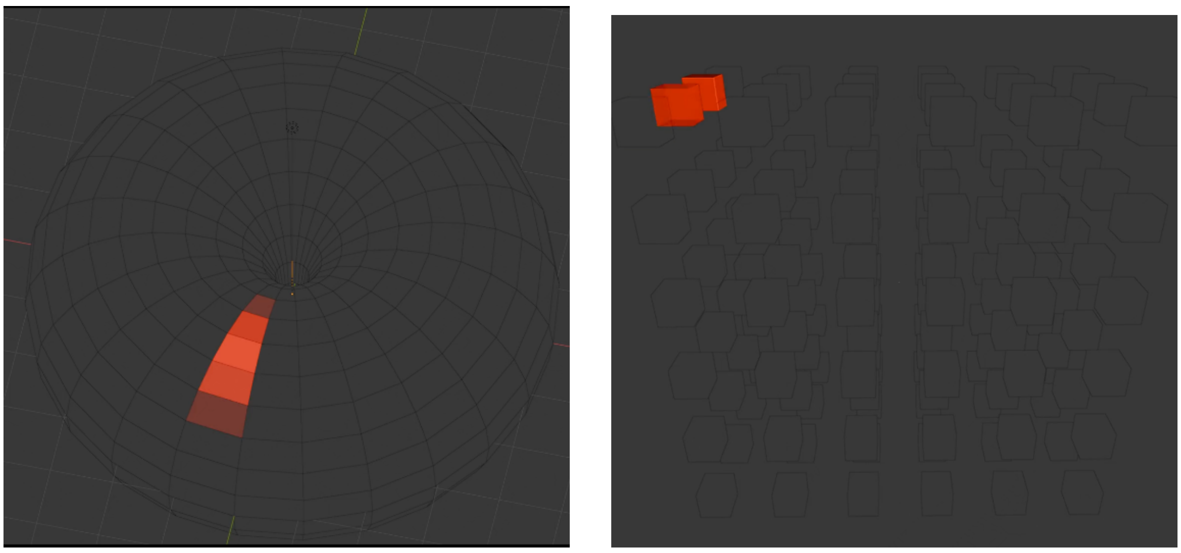


Figure 5.26: Spacing ranks and grouping CC-Numa domains together
Open in YouTube

5.4.6 Identifying the Ranks



(a) Torus
Open in YouTube

(b) Cube
Open in YouTube

Figure 5.27: identifying the ranks with sequential highlighting

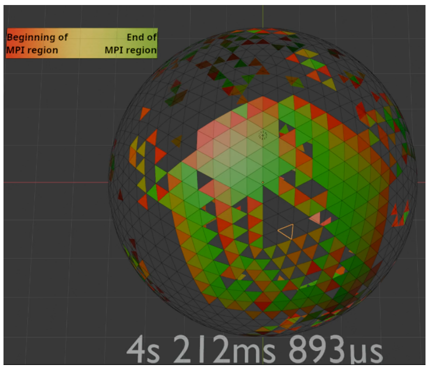
A crucial information for the user is to know which ranks corresponds to which cube or square. Labels are indeed an option, but they can bloat the display quite fast. We could simply label a few key ranks. But a quick and straightforward proposal was made with including as an introduction in the output video an animation: every rank from 0 to last lights up one after the other sequentially.

Evaluation

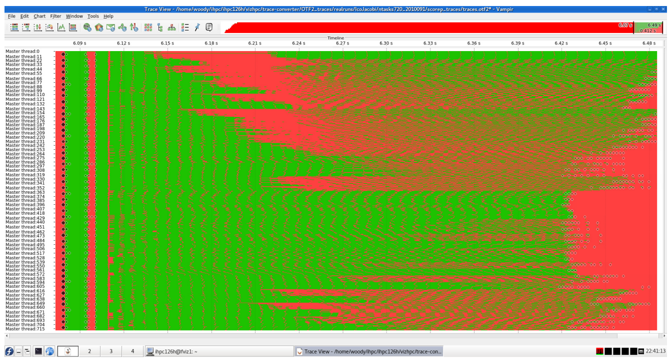
6.1 Effectiveness

Our visualizations displayed accurately the various trace data that we generated, and helped display idle waves straightforwardly, giving an interesting and useful addition to the traditional display. It also revealed some communication patterns and their effect on the execution, e.g. with the ranks sharing the same CC-Numa domain having priviledged communication, which showed immediately by forming lines in the Grid video.

One obvious added value is the ability to display complex domain in 3D. Figure 6.1, showing longside the Blender and Vampir visualizations for our Sphere case, is probably the best example of this. Not only is the unintuitive pinning easily understood, but in this particular screenshot, we can also see around the highlighted rank 5 a total of 17 ranks that are transparent, therefore outside of an MPI region. Immediately, this gives us an intuition that they belong to the same CC-Numa domain, which is correct, but very hard to see in the trace.



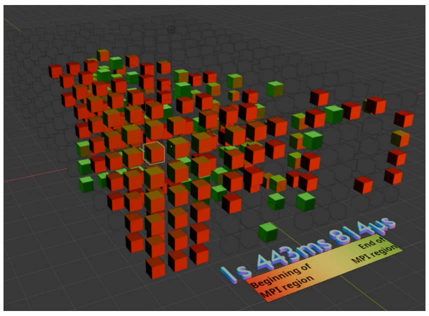
(a) Blender display
Open in YouTube



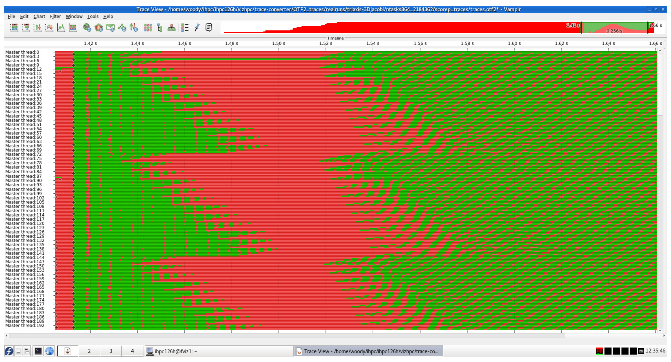
(b) Corresponding Vampir visualization

Figure 6.1: Run over 2880 ranks (40 nodes)

Another case, shown in Figure 6.2, is that of the cube, showing idle waves in the form of a euclidian plane, with vertical 6x3 CC-Numa domains. A quick glance at the Vampir visualization can show this. But we also notice that its direction changes over the run, and Vampir cannot accurately display in 2D all direction changes happening in 3D. Hence, this is accurately displayed only in Blender. Not only that, displaying the underlying pinning with regard to the CC-Numa domains gives some subtle insight into what is influencing that change: they are aligned along the Z-axis, and it is an interesting assumption that the faster communication along that axis provoked the corresponding rotation and switched the direction of the plane of the idle wave front.



(a) Blender display
Open in YouTube



(b) Corresponding Vampir visualization

Figure 6.2: Displayed CC-Numa domains in the cube run

6.2 Usability

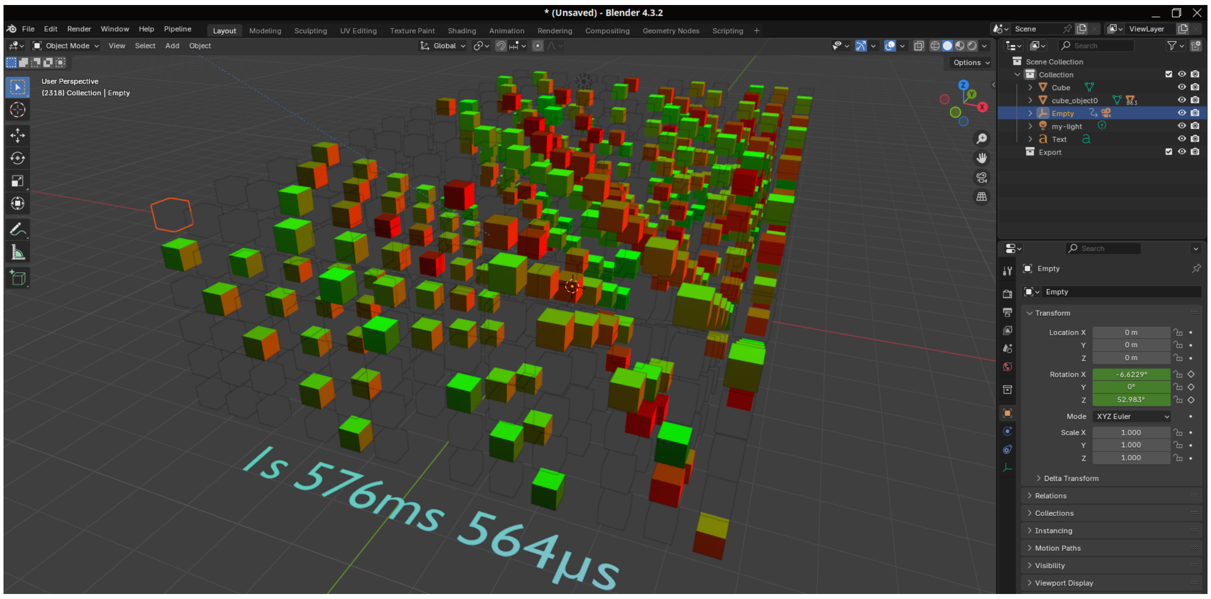
The Blender-based system required minimal setup and provided a qualitative 3D scene with a powerful and an effective Python API and GUI that were adaptable and provided high quality videos.

However, while this video generation is a good first objective to explore and assess the possibilities and their added value, it is also the most rigid and probably the worst output possible. Indeed, once rendered and saved, the only modification that the user can bring is by interacting with the progress bar, and even then, the control in a player such as VLC is very rigid and coarse, mostly skipping the in-between frames when moving its cursor from time A to time B, while YouTube does even worse by intentionally blurring them in addition to the skipping. Moreover, moving frame-by-frame is only possible going forward in VLC, with the E shortcut, while YouTube has a frame-stepping options in both directions, through the point key and the comma key, that quickly needs buffering. They were simply not designed for those uses.

This adds to a video compression that isn't frame-perfect, being mostly designed to save space and bandwidth, they only store an image difference with the previous frame and need to be decoded. Outputting to a video could be compared to using a screenshot of Vampir.

However, as we demonstrated, controlling other parameters such as the view angle is not only practical, it can be crucial to see certain hidden behaviors, or even simply to see the ranks that we are interested in. Only video editors would propose a finely controlled progress bar over those videos, and this represent a significant investment compared to the meager reward.

Within Blender however, the scene is not rendered yet, and it stores not pixel but the original source data much like vectorial images (typically in the SVG format) store the algorithmic formula generating an image, rather than the list of pixels composing it like raster images (usually in PNG or JPEG formats), allowing us to change the zooming with no loss and with a light memory footprint. Therefore, an ideal tool would not necessarily render to a video, but rather allow the user to move through a scene and select zoom, angle, and other display characteristics on-the-fly through simple and intuitive mouse and trackpad movements. Alternatively, the 3-axis figure on the top-right corner of the view can serve the same purpose. It is also easy to go over the frame, by simply hovering the current frame number on the lower-right corner, and scrolling more or less quickly to go over any particular set of frames.


* (Unsaved) - Blender 4.3.2
File Edit Render Window Help Pipeline
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Geometry Nodes Scripting
Object Mode View Select Add Object
Global
Options
User Perspective
(2318) Collection | Empty
Scene ViewLayer
Scene Collection
Collection
Cube
cube_object0
Empty
my-light
Text
Export
Is 576ms 564µs

Empty

Transform
Location X 0 m
Y 0 m
Z 0 m
Rotation X -6.6229°
Y 0°
Z 52.983°
Mode XYZ Euler
Scale X 1.000
Y 1.000
Z 1.000
Delta Transform
Relations
Collections
Instancing
Motion Paths
Visibility
Viewport Display

Figure 6.3: The Blender display and controls

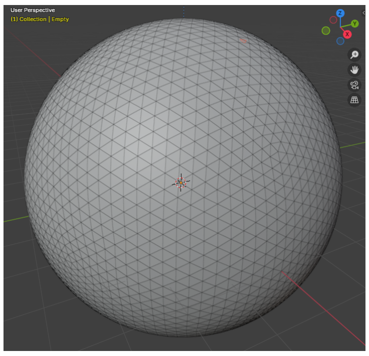
6.3 Scalability

6.3.1 Trace Extraction Script

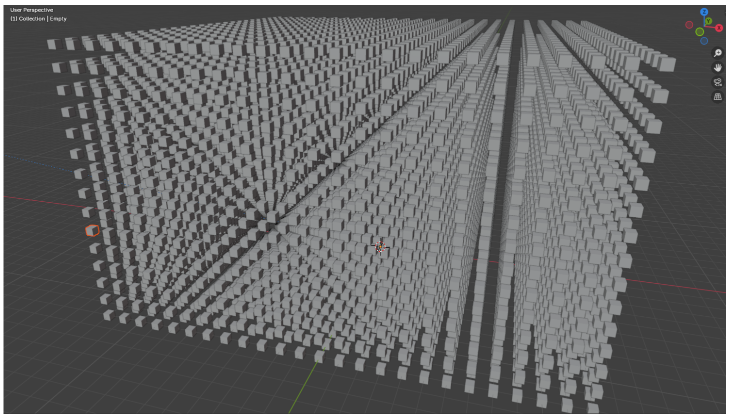
Using the OTF2 Python package, this script was both stable and versatile, as long as the trace file was light and short. To make it more scalable, we used common methods such as using the Numpy package and pre-allocating the memory that is later populated with the extracted trace data. We then used more complicated loops to read the trace only once after that, instead of looping over the ranks. This allowed us a roughly 100 times faster execution. However, using a standard office laptop, Blender would not handle more than 100MB worth of trace files. To lower the trace space on disk, we also implemented outputting the extracted timestamps as offsets, and to a binary file. This roughly halves the output file size.

6.3.2 Blender's Limitations

Understandably, running our Blender script was always the main problem, as we experienced freezes and many crashes. Generating objects alone could scale fairly well on our laptop, scaling to up to 10000 objects, generated within 10 seconds. After that,



(a) 8000 ranks Sphere



(b) 25x25x15 (9375) ranks Cube

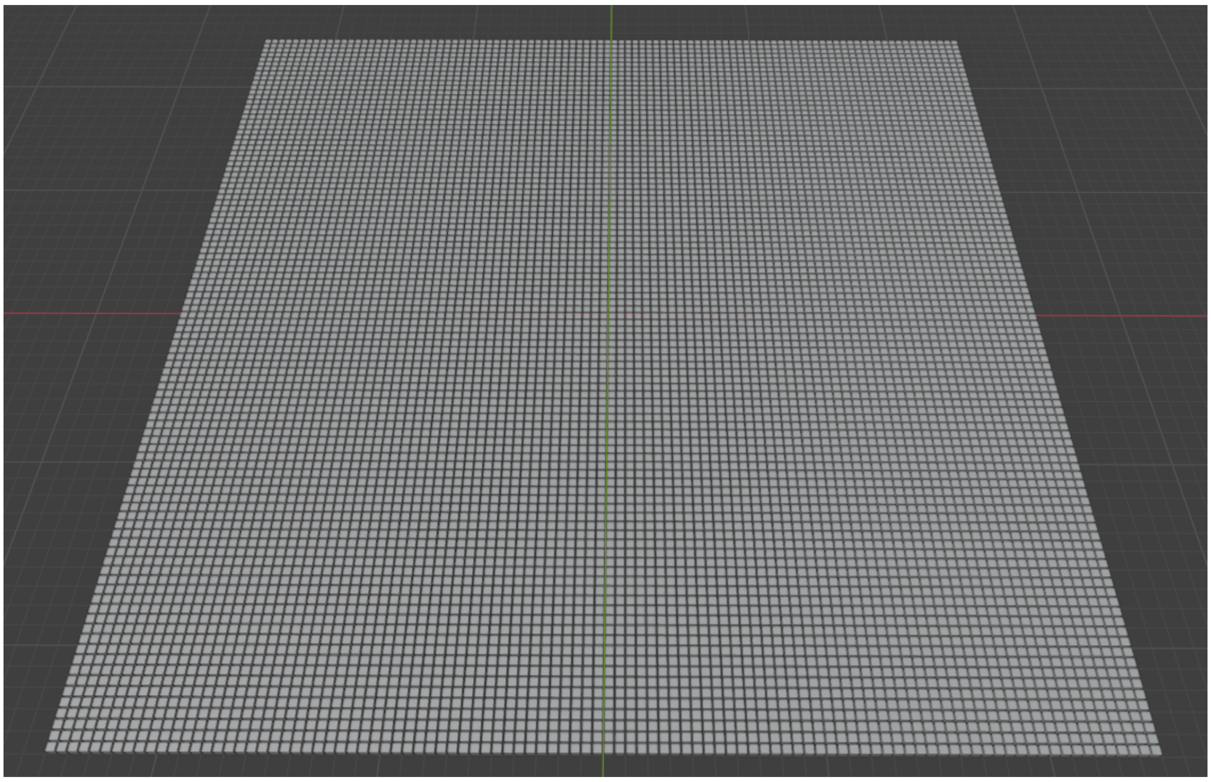


Figure 6.5: 100x100 ranks 2D grid

we needed to color the objects along the frames. This scaled up to mid-sized jobs of around 1500 processes before Blender started slowing down or freezing. We managed to display 2880 ranks, therefore 40 nodes, with the sphere, but since the generated objects are only triangles instead of cubes, they are less heavy on the memory. The same freezes and crashes were also experienced when generating a high number of frames, typically over 100000. A more powerful computer would indeed be able to handle significantly higher workloads, but that is exactly what a tool would do well to avoid, finding and implementing a lightweight solution.

Future Work and Possible Directions

- Adding all the standard capabilities, such as collective operations events, or OpenMP events would be the first addition, in order to get a complete tool handling all basic situations.
- Support for VR-based MPI trace inspection for easily and a ”true” 3D display instead of a 2D screen with identical input in both eyes.
- Going away from Blender: using Javascript to display over the browser though an integration with real-time viewers such as Unity or WebGL. This would allow to interactively change the view angle and the zoom, as well as allow specific behavior such as displaying information at hovering or when selecting objects, different color schemes changeable on-the-fly, switching the display to show other data such as the iteration number or the load imbalance...
- Developing the ability to modify the display speed, i.e. being able to change its cycle jump per frame, which would make the user go over the trace slower when fine detail and granularity are needed or faster when a more global insight is preferred, with an easy and intuitive action such as a simple mouse scrolling.
- Making it a live-tool able to inspect an ongoing execution. This could potentially be combined with the previous feature.
- Improve the scaling, to handle both a great number of processes and trace size, probably by implementing an equivalent of the Vampir server.

Conclusion

This thesis, describing itself as an exploration, was merely a first step towards an innovative visualization method. It aimed at demonstrating the feasibility and usefulness of using a 3D visualization for displaying MPI trace data, evaluating each specific viewing method and assessing its added value. By mapping each rank to a corresponding object in a 3D scene, and then mapping each region to their specific rank, we could create a new visualization that offers enhanced interpretability, making it valuable for developers and researchers optimizing parallel applications.

Using our tool, we could develop a deeper insight into the codes we studied, compared to just using Vampir. Although many of the codes' behaviors were noticeable on the latter, we could have them truly stand-out, and sometimes even add some key details that would not be shown on Vampir, such as the idle wave front's direction shift on our 3D Jacobi code, and its possible link to the CC-Numa domain's disposition.

This was done through two key additions: showing the trace through an animation allowed us to free the x-axis that was formerly dedicated to time, and using a 3D scene gave us an additional dimension, therefore allowing us to display ranks arranged in the form of a cuboid.

Regarding the display scalability, another practical addition was that this 3D scene allowed us to see that a very high number of ranks, such as with a 3D grid of 25x25x15 ranks or a 2D grid of 100x100 ranks. Far from bloating the scene, they are still easily displayed in an understandable, and not particularly overwhelming view, when it would be a struggle with Vampir to usefully show all that at once. We actually did not go further simply because we were stopped by the low computational limits of memory and compute power to display them on an office laptop. These number are much lower when combining that display to the coloring of each rank along frames, where it is best to limit ourselves to a few thousands ranks.

Indeed, Blender being widely known for being prone to crashes and for its hunger for power and memory, it could not be satisfactory in the context of a long run development of a tool, and it seems preferable and reasonable to move away from Blender for a more stable and lightweight alternative, ideally not needing the user to install additional software, such as a Javascript implementation running within a browser, which would allow us many interesting additions such as specific behaviors with hovering or selecting objects.

Of course, the downside to that is that it would lead us to giving up outputting our view to a video, since there is no easy way to encode them, especially across browsers and in a standardized way. But this comes a very low price, indeed, they are the least interactive and most rigid possible output, and have virtually no added value for the user in the context of exploring a trace compared to viewing the trace within the 3D scene itself before it is rendered: nobody would prefer a Vampir outputting to a simple image.

While such a tool will not replace Vampir or other existing trace visualization tools, it does not propose to do so. But merely to be an interesting, useful and handy addition to it, allowing the user to explore deeper into the trace data, and to make it faster to grasp and study it.

Bibliography

[1] Ayesha Afzal, Georg Hager, and Gerhard Wellein. “Analytic Modeling of Idle Waves in Parallel Programs: Communication, Cluster Topology, and Noise Impact”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Bradford L. Chamberlain et al. Vol. 12728 LNCS. Springer Science and Business Media Deutschland GmbH, June 24, 2021, pp. 351–371. ISBN: 978-3-030-78712-7. DOI: 10.1007/978-3-030-78713-4_19.

[2] Ayesha Afzal, Georg Hager, and Gerhard Wellein. *DisCostiC: A DSL-based Parallel Simulation Framework using First-Principles Analytic Performance Models*. Series: PASC’22. June 27, 2022. URL: https://www.youtube.com/watch?v=RKEyaa_khcY.

[3] Ayesha Afzal, Georg Hager, and Gerhard Wellein. *DisCostiC: Simulating MPI Applications Without Executing Code*. Series: SC’24. Nov. 17, 2024. URL: https://sc24.supercomputing.org/proceedings/poster/poster_pages/post167.html.

[4] Ayesha Afzal, Georg Hager, and Gerhard Wellein. *DisCostiC:Digital Twin Performance Simulations Unlocking Hardware-Software Interplay*. Series: ISC’25. June 10, 2025. URL: https://isc.app.swapcard.com/widget/event/isc-high-performance-2025/planning/UGxhbm5pbmdfMjU4NDU1Mg==.

[5] Ayesha Afzal, Georg Hager, and Gerhard Wellein. *Physical Oscillator Model for Parallel Distributed Computing*. Series: ISC-HPC’21. June 24, 2021. URL: https://app.swapcard.com/event/isc-high-performance-2021-digital/planning/UGxhbm5pbmdfNDUzNTk2.

[6] Ayesha Afzal, Georg Hager, and Gerhard Wellein. “Propagation and Decay of Injected One-Off Delays on Clusters: A Case Study”. In: *Proceedings of the 2019 IEEE International Conference on Cluster Computing*. Vol. 2019-September. CLUSTER’19. Institute of Electrical and Electronics Engineers Inc., Sept. 23, 2019, pp. 1–10. ISBN: 978-1-7281-4734-5. DOI: 10.1109/CLUSTER.2019.8890995.

[7] Ayesha Afzal, Georg Hager, and Gerhard Wellein. “The Role of Idle Waves, Desynchronization, and Bottleneck Evasion in the Performance of Parallel Programs”. In: *IEEE Transactions on Parallel and Distributed Systems, TPDS* (2022). DOI: 10.1109/TPDS.2022.3221085.

[8] *Blender 4.3 - Python API*. URL: https://docs.blender.org/api/4.3/.

[9] *Fritz - Documentation*. URL: https://doc.nhr.fau.de/clusters/fritz/.

[10] *OTF2 3.0.3 - HTML Manual*. URL: https://perftools.pages.jsc.fz-juelich.de/cicd/otf2/doc.7f6882e3/.

[11] *OTF2 3.0.3 - PDF Manual.* URL: https://perftools.pages.jsc.fz-juelich.de/cicd/otf2/doc.7f6882e3/python/index.html.

[12] *OTF2 3.0.3 - Python Interface Reference Guide.* URL: https://perftools.pages.jsc.fz-juelich.de/cicd/otf2/doc.7f6882e3.pdf.

[13] Lucas Mello Schnorr, Guillaume Huard, and Philippe O. A. Navaux. "Triva: Interactive 3D visualization for performance analysis of parallel applications". In: *Future Generation Computer Systems* 26.3 (2010), pp. 348–358. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2009.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0167739X09001563.

[14] *Score-P - Cheat Sheet.* URL: https://vampir.eu/public/files/pdf/spcheatsheet_a4.pdf.

[15] *Score-P 8.1 - Documentation.* URL: https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/tags/scorep-8.1/html/.

Appendix: Used Codes

A.1 Batch Job Script

```
#!/bin/bash -l
#
#================================================================================#
#                            Usage of this script:                               #
#  Find double question marks, and replace them with the corresponding values   #
#          Some comments can be crucial to read for correct settings             #
#================================================================================#
#
#SBATCH --ntasks-per-socket=??
#SBATCH --sockets-per-node=??
#SBATCH --ntasks-per-node=??
#SBATCH --nodes=??
#SBATCH --ntasks=??
#SBATCH --cpus-per-task=1
#SBATCH --cpu-freq=2400000-2400000:performance
#SBATCH --time=00:??:00
#SBATCH --job-name=comment??-ntasks??-program_name??
#SBATCH --mail-type=TIME_LIMIT # or =ALL,TIME_LIMIT_80
# #SBATCH --mail-user=??@?.?
#SBATCH -o ../ntasks??_%j/sbatch_output/out-%x-%j-on-%N.out
#SBATCH -e ../ntasks??_%j/sbatch_output/err-%x-%j-on-%N.err
#SBATCH --export=NONE         # don't export env from submitting shell
# First non-comment line ends SBATCH options, so the next one does




#=========================================#
# Preparing run environment and folders #
#=========================================#

# enable export of environment from this script to srun
unset SLURM_EXPORT_ENV

# Setup job environment (load modules, stage data, ...)
```

```
module load scorep

# Comment if you don't want this script to handle the compilation
make distclean
make -j
make clean

# Export the results folder name for readability
export RESULTS_FOLDERNAME=ntasks$SLURM_NTASKS\_$SLURM_JOB_ID

# Score-P tracing parameters
export SCOREP_ENABLE_TRACING=true
export SCOREP_TOTAL_MEMORY=3GB #Default is 16M

# Create the necessary directories and related variables to save useful info
cd ..
export TRACE_CONVERTER_ROOT_FOLDER=$PWD/$RESULTS_FOLDERNAME/
export PROGRAM_NAME=$(basename $(pwd))
mkdir -p $RESULTS_FOLDERNAME/extract/
mkdir -p $RESULTS_FOLDERNAME/sbatch_output/

# Save this job script to the sbatch_output folder in case we need to debug
cp Code/job??.sh $RESULTS_FOLDERNAME/sbatch_output/job.sh

# Save the used code to the sbatch_output folder in case we need to debug
mkdir -p $RESULTS_FOLDERNAME/sbatch_output/used_code/
cp Code/?? $RESULTS_FOLDERNAME/sbatch_output/used_code/

# Come back to the Code folder
cd Code

# OpenMP and Hybrid jobs parameters
export OMP_NUM_THREADS=1              # 1 OpenMP thread per MPI process
# for OpenMP, set number of threads to requested cpus-per-task
#export OMP_NUM_THREADS=£SLURM_CPUS_PER_TASK
# for Hybrid Job, for Slurm version >22.05: cpus-per-task has to be set again
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK




#=======================================#
# Printing program parameters           #
#=======================================#

# ###-commented ones don't work/exist?
echo "=== PROGRAM PARAMETERS: ================================================="
echo
```

```
echo "=== SLURM_CLUSTER_NAME      = "$SLURM_CLUSTER_NAME
echo "=== SLURM_JOB_PARTITION     = "$SLURM_JOB_PARTITION
echo "=== SLURM_JOB_NODELIST      = "$SLURM_JOB_NODELIST
echo "=== SLURM_JOB_ID            = "$SLURM_JOB_ID
echo "=== SLURM_NTASKS_PER_SOCKET = "$SLURM_NTASKS_PER_SOCKET
echo "=== SLURM_SOCKETS_PER_NODE  = "$SLURM_SOCKETS_PER_NODE ###
echo "=== SLURM_NTASKS_PER_NODE   = "$SLURM_NTASKS_PER_NODE
echo "=== SLURM_JOB_NUM_NODES     = "$SLURM_JOB_NUM_NODES
echo "=== SLURM_NTASKS            = "$SLURM_NTASKS
echo "=== SLURM_CPUS_PER_TASK     = "$SLURM_CPUS_PER_TASKSRUN_CPUS_PER_TASK
echo "=== SRUN_CPUS_PER_TASK      = "$SRUN_CPUS_PER_TASK
echo "=== OMP_NUM_THREADS         = "$OMP_NUM_THREADS
echo "=== SLURM_CPU_FREQ_REQ      = "$SLURM_CPU_FREQ ###
echo "=== SBATCH_TIMELIMIT        = "$SBATCH_TIMELIMIT ###
echo "=== SLURM_JOB_NAME          = "$SLURM_JOB_NAME
echo "=== SBATCH_OUTPUT           = "$SLURM_OUTPUT ###
echo "=== SBATCH_EXPORT           = "$SBATCH_EXPORT ###
echo "=== SLURM_CPU_BIND_VERBOSE  = "$SLURM_CPU_BIND_VERBOSE ###
echo "=== SLURM_CPU_BIND_TYPE     = "$SLURM_CPU_BIND_TYPE ###
echo "=== SLURM_CPU_BIND_LIST     = "$SLURM_CPU_BIND_LIST ###
echo "=== SLURM_CPU_BIND          = "$SLURM_CPU_BIND ###
echo
echo "=== SCOREP_ENABLE_TRACING       = "$SCOREP_ENABLE_TRACING
echo "=== SCOREP_TOTAL_MEMORY         = "$SCOREP_TOTAL_MEMORY
echo "=== SCOREP_FILTERING_FILE       = "$SCOREP_FILTERING_FILE
echo "=== SCOREP_METRIC_PERF          = "$SCOREP_METRIC_PERF
echo "=== SCOREP_METRIC_PAPI          = "$SCOREP_METRIC_PAPI
echo "=== SCOREP_EXPERIMENT_DIRECTORY = "$SCOREP_EXPERIMENT_DIRECTORY
echo
echo "=== TRACE_CONVERTER_ROOT_FOLDER = "$TRACE_CONVERTER_ROOT_FOLDER
echo "=== RESULTS_FOLDERNAME          = "$RESULTS_FOLDERNAME
echo "=== SLURM_SUBMIT_DIR            = "$SLURM_SUBMIT_DIR
echo "=== PROGRAM_NAME                = "$PROGRAM_NAME
echo "=== COMMENTS                    = ??" # put comments here, filtered?
echo "=== COMMENTS                    = " # input file/problem size
echo "=== COMMENTS                    = " # code modifications?
echo "=== COMMENTS                    = " # compilation and run conditions?
echo "=== COMMENTS                    = " # omp threads/deactivated?
echo
echo "=== Running the program now!"
echo
echo



#===================#
# Run command       #
```

```
#===================#

# Run command with normal binding
srun --verbose --cpu-freq=2400000-2400000:performance \
--cpu-bind=verbose,cores ./binary?? # < input #??




#=======================================#
# Python : Extraction of trace data    #
#=======================================#

module load python
conda activate trace
python ../../../../trace_converter.py --root_folder=$TRACE_CONVERTER_ROOT_FOLDER
```

A.2 Makefile

```
CC = scorep mpiicc
# to have compilation time filtering, replace with:
#CC = scorep --instrument-filter=./scorep_filter_file mpiicc

FLAGS_OPENMP = -qopenmp
FLAGS_DEBUG = -g -O0 # -g3 includes extra info such as macro expansions
FLAGS_FAST = -O3 -xAVX -fno-alias
FLAGS_WARNING = -std=c11 -Wall -Wextra -pedantic
FLAGS_PROFILING = -pg
FLAGS_MACROS = -DREAD_INPUT #-MMD -MP
INCLUDES =
LIBS = -lm # aka LDFLAGS

FLAG_LIST_BUILD = $(FLAGS_MACROS) $(FLAGS_WARNING) $(FLAGS_FAST)
FLAG_LIST_DEBUG = $(FLAGS_MACROS) $(FLAGS_WARNING) $(FLAGS_DEBUG)

DEBUG = No
ifeq ($(DEBUG),No)
CFLAGS = $(FLAG_LIST_BUILD)
LIBS = -lm
else
CFLAGS = $(FLAG_LIST_DEBUG)
LIBS = -lm
endif

SRCDIR = .
SRC = $(wildcard $(SRCDIR)/*.c)
OBJ = $(SRC:.c=.o)
```

```
EXEC = binary


all: distclean $(EXEC) clean


binary: $(OBJ)
@$(CC) $(CFLAGS) -o $@ $^ $(LIBS)
@echo "\033[1m"
@echo "   DEBUG was set to $(DEBUG)."
@echo "   Compilation done."
@echo "\033[0m"


# In case there is a .h file, the dependencies are not rebuilt
# automatically when it is modified unless we add the following:
#main.o: filename.h
main.o: proc_info.h
solver.o: proc_info.h

%.o: $(SRCDIR)/%.c
@$(CC) $(CFLAGS) -c $< -o $@


clean:
@rm -rf *.o
#        @rm -rf *.o 2> /dev/null # Suppresses error messages

distclean:
@rm -rf $(EXEC) *.o
#        @rm -rf £(EXEC) *.o 2> /dev/null # Suppresses error messages

# If there is a file named clean newer than its dependencies in the
# directory, it will not be executed unless we add the following:
.PHONY: distclean #dependencies will be systematically rebuilt
```

A.3 Optional Score-P Filtering File

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE *
SCOREP_REGION_NAMES_END
```

A.4 Triaxis-3Djacobi - Main

```
/*
* Copyright (c) 2021, Dirk Pleiter, KTH
*
```

```
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*  * Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*  * Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*  * Neither the name of NVIDIA CORPORATION nor the names of its
*    contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
* OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <limits.h>

#include <mpi.h>
// #include <hwloc.h>
// #include <sched.h>
#include "proc_info.h"

//#ifndef MEASURETIME
//#define MEASURETIME 0
//#endif

/*
* Parallel solver implemented in the solver.c file
*/
int solver(double *, double *, int, int, int, double, int, \
struct proc_info *, int, int, int, int, int, int);

/*
* Helper function that calculates the optimal partitioning of processes
```

```
for the current domain.
*/
static void find_optimal_partitioning(int nx, int ny, int nz, int size, \
int *NPROCX, int *NPROCY, int *NPROCZ);


int main(int argc, char** argv)
{

        /*
        * Setup Phase
        */
        /*Initialize MPI and the process info struct*/
        MPI_Init(&argc, &argv);
        struct proc_info proc;

        MPI_Comm_size(MPI_COMM_WORLD, &proc.size);

        /*Get run parameters from input file and communicate it to all ranks*/

        int myrank;
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

        int NX, NY, NZ, NPROCX, NPROCY, NPROCZ, PERIODICITY_X, PERIODICITY_Y, \
        PERIODICITY_Z, NMAX, INJECTION_ITERATION, WORKLOAD_ITERATIONS, LAGGER_RANK;
        double EPS;

        if (myrank == 0)
        {
                /*Defaults*/
                NX = 455;
                NX += 2; // Add 2 for boundary points
                NY = 455;
                NY += 2; // Add 2 for boundary points
                // NZ = 455;
                // NZ += 2; // Add 2 for boundary points
                PERIODICITY_X = 0;      // Setting periodicity to 0 (= False)
                PERIODICITY_Y = 0;
                PERIODICITY_Z = 0;
                INJECTION_ITERATION = 3;
                LAGGER_RANK = 5;
                WORKLOAD_ITERATIONS = 100000000;
                NMAX = 40;
                EPS = 1e-5;
                #ifdef READ_INPUT
                printf("\nInput NX, the grid size in the x direction:");
                scanf("%d", &NX);
                NX += 2; // Add 2 for boundary points
```

```
printf("\nInput NY, the grid size in the y direction:");
scanf("%d", &NY);
NY += 2; // Add 2 for boundary points
// printf("\nInput NZ, the grid size in the z direction:");
// scanf("%d", &NZ);
// NZ += 2; // Add 2 for boundary points
// printf("\nInput NPROCX, number of ranks in the x direction:");
// scanf("%d", &NPROCX);
// printf("\nInput NPROCY, number of ranks in the y direction:");
// scanf("%d", &NPROCY);
// printf("\nInput NPROCZ, number of ranks in the z direction:");
// scanf("%d", &NPROCZ);
printf("\nInput PERIODICITY_X, 0 for false:");
scanf("%d", &PERIODICITY_X);
printf("\nInput PERIODICITY_Y, 0 for false:");
scanf("%d", &PERIODICITY_Y);
printf("\nInput PERIODICITY_Z, 0 for false:");
scanf("%d", &PERIODICITY_Z);
printf("\nInput INJECTION_ITERATION, the iteration number for \
the extra workload injection:");
scanf("%d", &INJECTION_ITERATION);
printf("\nInput LAGGER_RANK, the rank of the lagger for the \
extra worload injection:");
scanf("%d", &LAGGER_RANK);
printf("\nInput WORKLOAD_ITERATIONS, the iteration number for \
the extra workload:");
scanf("%d", &WORKLOAD_ITERATIONS);
printf("\nInput NMAX, the maximum iterations for the solver:");
scanf("%d", &NMAX);
printf("\nInput EPS, the error tolerance for the solver:");
scanf("%lf", &EPS);
#endif
NZ = (int) ceil((20 * 1024 * 1024 / (8*2)) * proc.size / \
((NX - 2) * (NY - 2)));
NZ += 2; // Add 2 for boundary points
find_optimal_partitioning(NX-2, NY-2, NZ-2, proc.size, \
&NPROCX, &NPROCY, &NPROCZ);
printf("\n-> Total grid size: NX=%d x NY=%d x NZ=%d"
"\n-> Local grid size: NPROCX=%d x NPROCY=%d x NPROCZ=%d"
"\n-> Periodicity in X, Y and Z is: %d x %d x %d (0 is False)"
"\n-> Injecting an extra workload on iteration %d on rank %d"
"\n-> Extra workload is %d iterations"
"\n-> Max number of iterations: %d"
"\n-> Tolerance: %f\n\n",
NX, NY, NZ,
NPROCX, NPROCY, NPROCZ,
PERIODICITY_X, PERIODICITY_Y, PERIODICITY_Z,
INJECTION_ITERATION, LAGGER_RANK,
        WORKLOAD_ITERATIONS,
        NMAX,
        EPS);
}


/* Send input parameters to all procs */
MPI_Bcast(                  &NX, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(                  &NY, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(                  &NZ, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(              &NPROCX, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(              &NPROCY, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(              &NPROCZ, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(       &PERIODICITY_X, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(       &PERIODICITY_Y, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(       &PERIODICITY_Z, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(                &NMAX, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(&INJECTION_ITERATION, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(         &LAGGER_RANK, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(&WORKLOAD_ITERATIONS, 1, MPI_INT,    0, MPI_COMM_WORLD);
MPI_Bcast(                 &EPS, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/*Set-up distribution of the grid among a 3D process arrangement*/
proc.dims[1] = NPROCX;
proc.dims[0] = NPROCY;
proc.dims[2] = NPROCZ;

/*Set up the cartesian communicator with reordering set to 0*/
int periods[3] = {PERIODICITY_X,PERIODICITY_Y,PERIODICITY_Z};
MPI_Cart_create(MPI_COMM_WORLD, 3, proc.dims, periods, 0, &proc.cartcomm);

MPI_Comm_rank(proc.cartcomm, &proc.rank);

/*Get the ranks of the neighboring processes in all directions*/
MPI_Cart_shift(proc.cartcomm, 0, 1, &proc.neighbors[BACK], \
&proc.neighbors[FRONT]);
MPI_Cart_shift(proc.cartcomm, 1, 1, &proc.neighbors[LEFT], \
&proc.neighbors[RIGHT]);
MPI_Cart_shift(proc.cartcomm, 2, 1, &proc.neighbors[DOWN], \
&proc.neighbors[UP]);

/*Get own cartesian coordinates*/
MPI_Cart_coords(proc.cartcomm, proc.rank, 3, proc.coords);

/*Local size of the grid; Additional rows and columns for boundary
points*/
size_t local_nx = ((NX-2)/proc.dims[1]) + 2;
size_t local_ny = ((NY-2)/proc.dims[0]) + 2;
size_t local_nz = ((NZ-2)/proc.dims[2]) + 2;

int y_offset = (local_ny - 2) * proc.coords[0];
int x_offset = (local_nx - 2) * proc.coords[1];
int z_offset = (local_nz - 2) * proc.coords[2];

/*
* If the grid points are not divisible by the processor dimension.
* The last processor on the axis will receive the remaining points.
*/
if (proc.coords[0] == proc.dims[0] - 1)
{
        local_ny += ((NY - 2) % proc.dims[0]);
}

if (proc.coords[1] == proc.dims[1] - 1)
{
        local_nx += ((NX - 2) % proc.dims[1]);
}

if (proc.coords[2] == proc.dims[2] - 1)
{
        local_nz += ((NZ - 2) % proc.dims[2]);
}


// hwloc_topology_t topology;
// int nbcores;
//
// hwloc_topology_init(&topology);  // initialization
// hwloc_topology_load(topology);   // actual detection
//
// nbcores = hwloc_get_nbobjs_by_type(topology, HWLOC_OBJ_CORE);
// printf("%d cores\n", nbcores);
//
// //int last_cpu_location = 0;
// hwloc_bitmap_t set = hwloc_bitmap_alloc();
// //hwloc_set_cpubind(topology, set, HWLOC_CPUBIND_THREAD);
//
// int last_cpu_location = 0;//hwloc_get_last_cpu_location(topology, \
set, HWLOC_CPUBIND_THREAD); //0); \
//HWLOC_CPUBIND_PROCESS); //HWLOC_CPUBIND_THREAD)
// int cpubind = hwloc_get_cpubind(topology, set, \
HWLOC_CPUBIND_PROCESS); //0); //HWLOC_CPUBIND_PROCESS); //HWLOC_CPUBIND_THR
// //int proc_cpubind = hwloc_get_proc_cpubind(topology, set, pid_t, HWLOC_C
//HWLOC_CPUBIND_PROCESS); //HWLOC_CPUBIND_THREAD)
//
// int sched_cpu = sched_getcpu();
// unsigned int cpu, node;
// getcpu(&cpu, &node);

/*Print for each rank its coordinates, neighbors,
and other topology info*/
printf("\n-> My rank is: %d"
"\n-> My cartesian coordinates (x,y,z) are: (%d,%d,%d) for a \
total ranks number of (%d,%d,%d)"
"\n-> My neighbors are: RIGHT=%d, FRONT=%d, LEFT=%d, BACK=%d, \
UP=%d, DOWN=%d"
"\n-> My local grid size is: local_nx=%d, local_ny=%d, \
local_nz=%d for a total grid size of NX=%d, NY=%d, NZ=%d\n\n",
//"\n-> last_cpu_location=%d, cpubind=%d, sched_cpu=%d, cpu=%d, \
node=%d\n\n",
//"\n-> sched_cpu=%d, cpu=%d, node=%d\n\n",
proc.rank,
proc.coords[1], proc.coords[0], proc.coords[2],
proc.dims[1], proc.dims[0], proc.dims[2],
proc.neighbors[RIGHT],proc.neighbors[FRONT],proc.neighbors[LEFT],
proc.neighbors[BACK],proc.neighbors[UP],proc.neighbors[DOWN],
(int) local_nx, (int) local_ny, (int) local_nz, NX, NY, NZ);//,
//sched_cpu, cpu, node);
//last_cpu_location, cpubind, sched_cpu, cpu, node);

/*MPI Datatype for the communication of the boundary points
between processes*/
MPI_Type_vector(local_nz-2, local_nx-2, local_nx*local_ny,
        MPI_DOUBLE, &proc.roll_xz);
MPI_Type_commit(&proc.roll_xz);

// Using MPI_Type_create_struct instead of this non-fonctionning \
combination
// MPI_Type_vector(local_ny-2, 1, local_nx, MPI_DOUBLE, &proc.column);
// MPI_Type_commit(&proc.column);
// MPI_Type_vector(local_nz-2, 1, local_nx*local_ny, proc.column, \
&proc.pitch_yz);
// MPI_Type_commit(&proc.pitch_yz);


int *lengths = \
malloc((local_ny-2)*(local_nz-2)*sizeof(int));
MPI_Aint *displacements = \ malloc((local_ny-2)*(local_nz-2)*sizeof(MPI_Aint)
MPI_Datatype *types = \ malloc((local_ny-2)*(local_nz-2)*sizeof(MPI_Datatype)
for (int iz = 0; iz < local_nz-2; iz++)
for (int iy = 0; iy < local_ny-2; iy++)
{
        lengths[(local_ny-2)*iz + iy] = 1;
        displacements[(local_ny-2)*iz + iy] = (MPI_Aint) \
        ((local_nx*local_ny*(iz+1) + local_nx*(iy+1))*sizeof(double));
        types[(local_ny-2)*iz + iy] = MPI_DOUBLE;
}
MPI_Type_create_struct((local_nz-2)*(local_ny-2), lengths, \
displacements, types, &proc.pitch_xz);
MPI_Type_commit(&proc.pitch_yz);

MPI_Type_vector(local_nx-2, local_ny-2, local_nx, MPI_DOUBLE, \
&proc.yaw_xy);
MPI_Type_commit(&proc.yaw_xy);

/*
* End of setup phase
*/

double *v;
double *f;

// Allocate memory
v = (double *) malloc(local_nz * local_ny * local_nx * sizeof(double));
f = (double *) malloc(local_nz * local_ny * local_nx * sizeof(double));

// Initialise input
for (int iz = 0; iz < (int) local_nz; iz++)
for (int iy = 0; iy < (int) local_ny; iy++)
for (int ix = 0; ix < (int) local_nx; ix++)
{
        v[local_ny*local_nx*iz + local_nx*iy + ix] = 0.0;

        const double x = 2.0 * (ix+x_offset) / (NX - 1.0) - 1.0;//?
        const double y = 2.0 * (iy+y_offset) / (NY - 1.0) - 1.0;//?
        const double z = 2.0 * (iz+z_offset) / (NZ - 1.0) - 1.0;//?
        f[local_ny*local_nx*iz + local_nx*iy + ix] = sin(x + y + z);//?
}

/*Start timer*/
//#if MEASURETIME
//struct timespec ts;
//double start, end;
//if(proc.rank == 0)
//{
        //    clock_gettime(CLOCK_MONOTONIC, &ts);
        //    start = (double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9;
        //}
//#endif


// Call solver
solver(v, f, local_nx, local_ny, local_nz, EPS, NMAX, &proc, \
NX, NY, NZ, INJECTION_ITERATION, LAGGER_RANK, WORKLOAD_ITERATIONS);



/*End timer*/
//#if MEASURETIME
//if(proc.rank == 0)
//{
        //    clock_gettime(CLOCK_MONOTONIC, &ts);
        //    end = (double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9;
        //    printf("Execution time: %f s\n", end-start);
        //}
//#endif

//for (int iy = 0; iy < NY; iy++)
//    for (int ix = 0; ix < NX; ix++)
//        printf("%d,%d,%e\n", ix, iy, v[iy*NX+ix]);

// Clean-up
free(v);
free(f);

// hwloc_bitmap_free(set);
// hwloc_topology_destroy(topology);

MPI_Type_free(&proc.roll_xz);
MPI_Type_free(&proc.pitch_yz);
free(displacements);
free(types);
MPI_Type_free(&proc.yaw_xy);
//MPI_Type_free(&proc.column);
MPI_Finalize();

return 0;
}

static void find_optimal_partitioning(int nx, int ny, int nz, int size, \
int *NPROCX, int *NPROCY, int *NPROCZ)
{
int min_surface_area = INT_MAX;

// Try all factor combinations nprocx * nprocy * nprocz = size
for (int nprocx = 1; nprocx <= size; nprocx++)
if (size % nprocx == 0)
{
        int remaining_ranks = size / nprocx;
        for (int nprocy = 1; nprocy <= remaining_ranks; nprocy++)
        if (remaining_ranks % nprocy == 0)
        {
                int nprocz = remaining_ranks / nprocy;

                // Compute inter-rank surface area for this partitioning
                int area = nprocx*ny*nz + nprocy*nx*nz + nprocz*nx*ny;

                // Update minimum if better configuration found
                if (area < min_surface_area)
                {
                        min_surface_area = area;
                        *NPROCX = nprocx;
                        *NPROCY = nprocy;
                        *NPROCZ = nprocz;
                }
        }
}
}
```

A.5 Triaxis-3Djacobi - Proc_Info

```
#pragma once
#include <mpi.h>

#define FRONT 0
#define RIGHT 1
#define BACK 2
#define LEFT 3
#define UP 4
#define DOWN 5

struct proc_info {
        int rank;   /*Rank of the current process*/
        int size;   /*Number of processes in total*/
        int coords[3];  /*Coordinates of the current process in the grid*/
        int neighbors[6];   /*Neighbors of the process in the cartesian grid*/
        int dims[3];    /*Dimension of the cartesian grid*/
        MPI_Comm cartcomm; /*Cartesian communicator*/
        MPI_Datatype roll_xz, pitch_yz, yaw_xy, column; /*MPI Types for faces*/
        MPI_Request requests[12]; /*MPI Request handles instanciated in calls*/
        };
```

A.6 Triaxis-3Djacobi - Solver

```
/*
* Copyright (c) 2021, Dirk Pleiter, KTH
```

```
*
* This source code is in parts based on code from Jiri Kraus (NVIDIA) and
* Andreas Herten (Forschungszentrum Juelich)
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*  * Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*  * Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
*  * Neither the name of NVIDIA CORPORATION nor the names of its
*    contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND Any
* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
* CONTRIBUTORS BE LIABLE FOR Any DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON Any THEORY
* OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN Any WAY OUT OF THE USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "proc_info.h"

int solver(double *v, double *f, int nx, int ny, int nz, double eps,
int nmax, struct proc_info *proc, int NX, int NY, int NZ,
int INJECTION_ITERATION, int LAGGER_RANK, int WORKLOAD_ITERATIONS)
{
        int n = 0;
        double e = 2. * eps;
        double *vp;
        double sum = 1.0;

        vp = (double *) malloc(nx * ny * nz * sizeof(double));
```

```
MPI_Barrier(proc->cartcomm);



while ((n < nmax)) //&& (e > eps))
{
        e = 0.0;

        /*Computation Phase*/

        for( int iz = 1; iz < (nz-1); iz++ )
        for( int ix = 1; ix < (nx-1); ix++ )
        for (int iy = 1; iy < (ny-1); iy++)
        {
                double d;

                vp[ny*nx*iz     + nx*iy     + ix]   = -(1/6) *
                (f[ny*nx*iz + iy*nx + ix] - \
                (v[ny*nx*iz     + nx*iy     + ix+1] + \
                v[ny*nx*iz     + nx*iy     + ix-1] + \
                v[ny*nx*iz     + nx*(iy+1) + ix  ] + \
                v[ny*nx*iz     + nx*(iy-1) + ix  ] + \
                v[ny*nx*(iz+1) + nx*iy     + ix  ] + \
                v[ny*nx*(iz-1) + nx*iy     + ix  ]));

                d = fabs(vp[nx*ny*iz+nx*iy+ix] - v[nx*ny*iz+nx*iy+ix]);
                e = (d > e) ? d : e;
        }

        // Update v and compute error as well as error weight factor

        double w = 0.0;

        for(int iz = 1; iz < (nz-1); iz++)
        for(int ix = 1; ix < (nx-1); ix++)
        for (int iy = 1; iy < (ny-1); iy++)
        {
                v[ny*nx*iz + nx*iy + ix] = vp[ny*nx*iz + nx*iy + ix];
                w += fabs(v[ny*nx*iz + nx*iy + ix]);
        }

        /*End of computation phase*/

        /*Extra workload injection to generate an idle wave*/

        if ((n == INJECTION_ITERATION) && (proc->rank == LAGGER_RANK))
        {
                sum = 1.0;
                for (int j = 0; j < WORKLOAD_ITERATIONS; j++)
                sum = sum + 4.0 / (1.0 + j * j);

                printf("\n\n        Rank %d: Extra workload with a total \
                sum of %12.6f MBytes.\n\n", proc->rank, sum);
        }
        // {
                // e = 0.0;

                // /*Computation Phase*/
                // for( int iz = 1; iz < (nz-1); iz++ )
                //     for( int ix = 1; ix < (nx-1); ix++ )
                //         for (int iy = 1; iy < (ny-1); iy++)
                //         {
                        //              double d;

                        //              vp[ny*nx*iz      + nx*iy      + ix] \
                           = -(1/6) * (f[ny*nx*iz + iy*nx + ix] - \
                         //              (v[ny*nx*iz      + nx*iy      + ix+1]
                         + v[ny*nx*iz      + nx*iy      + ix-1] + \
                         //               v[ny*nx*iz      + nx*(iy+1) + ix   ]
                         + v[ny*nx*iz      + nx*(iy-1) + ix   ] + \
                         //               v[ny*nx*(iz+1) + nx*iy      + ix   ]
                         + v[ny*nx*(iz-1) + nx*iy      + ix   ]));

                        //              d = fabs(vp[nx*ny*iz+nx*iy+ix] - \
                         v[nx*ny*iz+nx*iy+ix]);
                        //              e = (d > e) ? d : e;
                        //         }

                // // Update v and compute error \
                 as well as error weight factor

                // double w = 0.0;

                // for(int iz = 1; iz < (nz-1); iz++)
                //     for(int ix = 1; ix < (nx-1); ix++)
                //         for (int iy = 1; iy < (ny-1); iy++)
                //         {
                        //              v[ny*nx*iz + nx*iy + ix] = \
                         vp[ny*nx*iz + nx*iy + ix];
                        //              w += fabs(v[ny*nx*iz + nx*iy + ix]);
                        //         }

                // /*End of computation phase*/

                //     //TODO: printf("\nRank %d: Extra workload with \
                 a total sum of %12.6f MBytes.\n", proc->rank, ?);
                // }

        /*End of injection phase*/

        /*Communication Phase*/

        MPI_Isend(&v[      1*ny*nx + (ny-2)*nx +       1], 1, proc->roll_xz,
        proc->neighbors[FRONT], 0, proc->cartcomm, &proc->requests[0]);
        MPI_Isend(&v[      1*ny*nx +       1*nx +       1], 1, proc->roll_xz,
        proc->neighbors[BACK],  0, proc->cartcomm, &proc->requests[1]);
        MPI_Isend(&v[      1*ny*nx +       1*nx + (nx-2)], 1, proc->pitch_yz,
        proc->neighbors[RIGHT], 0, proc->cartcomm, &proc->requests[2]);
        MPI_Isend(&v[      1*ny*nx +       1*nx +       1], 1, proc->pitch_yz,
        proc->neighbors[LEFT],  0, proc->cartcomm, &proc->requests[3]);
        MPI_Isend(&v[(nz-2)*ny*nx +       1*nx +       1], 1, proc->yaw_xy,
        proc->neighbors[UP],    0, proc->cartcomm, &proc->requests[4]);
        MPI_Isend(&v[      1*ny*nx +       1*nx +       1], 1, proc->yaw_xy,
        proc->neighbors[DOWN],  0, proc->cartcomm, &proc->requests[5]);

        MPI_Irecv(&v[      1*ny*nx + (ny-1)*nx +       1], 1, proc->roll_xz,
        proc->neighbors[FRONT], 0, proc->cartcomm, &proc->requests[6]);
        MPI_Irecv(&v[      1*ny*nx +       0*nx +       1], 1, proc->roll_xz,
        proc->neighbors[BACK],  0, proc->cartcomm, &proc->requests[7]);
        MPI_Irecv(&v[      1*ny*nx +       1*nx + (nx-1)], 1, proc->pitch_yz,
        proc->neighbors[RIGHT], 0, proc->cartcomm, &proc->requests[8]);
        MPI_Irecv(&v[      1*ny*nx +       1*nx +       0], 1, proc->pitch_yz,
        proc->neighbors[LEFT],  0, proc->cartcomm, &proc->requests[9]);
        MPI_Irecv(&v[(nz-1)*ny*nx +       1*nx +       1], 1, proc->yaw_xy,
        proc->neighbors[UP],    0, proc->cartcomm, &proc->requests[10]);
        MPI_Irecv(&v[      0*ny*nx +       1*nx +       1], 1, proc->yaw_xy,
        proc->neighbors[DOWN],  0, proc->cartcomm, &proc->requests[11]);

        MPI_Waitall(12, proc->requests, MPI_STATUSES_IGNORE);


        /*End of communication phase*/

        /*Compute weight on the boundary*/
        // This is cancelled because we don't actually care about the \
         Jacobi result and we want to have all computational load equal


        /*        if (proc->coords[0] == 0)
         {
                 for (int iz = 1; iz < (nz-1); iz++)
                 for (int ix = 1; ix < (nx-1); ix++)
                 {
                         //v[nx*1      + ix] = v[nx*0     + ix];
```

```
                        w += fabs(v[ny*nx*iz + nx*0 + ix]);
                }
        }

        if(proc->coords[0] == proc->dims[0]-1)
        {
                for (int iz = 1; iz < (nz-1); iz++)
                for (int ix = 1; ix < (nx-1); ix++)
                {
                        //v[nx*(ny-2) + ix] = v[nx*(ny-1)      + ix];
                        w += fabs(v[ny*nx*iz + nx*(ny-1) + ix]);
                }
        }

        if(proc->coords[1] == 0)
        {
                for (int iz = 1; iz < (nz-1); iz++)
                for (int iy = 1; iy < (ny-1); iy++)
                {
                        //v[nx*iy + 1]      = v[nx*iy + 0];
                        w += fabs(v[ny*nx*iz + nx*iy + 0]);
                }
        }

        if(proc->coords[1] == proc->dims[1]-1)
        {
                for (int iz = 1; iz < (nz-1); iz++)
                for (int iy = 1; iy < (ny-1); iy++)
                {
                        //v[nx*iy + (nx-2)] = v[nx*iy + (nx-1)];
                        w +=  fabs(v[ny*nx*iz + nx*iy + (nx-1)]);
                }
        }

        if(proc->coords[2] == 0)
        {
                for (int iy = 1; iy < (ny-1); iy++)
                for (int ix = 1; ix < (nx-1); ix++)
                {
                        //v[nx*ny*iz + 1]      = v[nx*ny*iz + 0];
                        w += fabs(v[ny*nx*0 + nx*iy + ix]);
                }
        }

        if(proc->coords[2] == proc->dims[2]-1)
        {
                for (int iy = 1; iy < (ny-1); iy++)
                for (int ix = 1; ix < (nx-1); ix++)
```

```
                        {
                                //v[nx*ny*iz + (ny-2)] = v[nx*ny*iz + (ny-1)];
                                w += fabs(v[ny*nx*(nz-1) + nx*iy + ix]);
                        }
                }
                */
                //TODO: Reduce to 1 reduction operation
                // We comment the Allreduces to avoid synchronization
                //and to make the idle wave appear in the trace
                //MPI_Allreduce(MPI_IN_PLACE, &e, 1, MPI_DOUBLE, MPI_MAX, \
                proc->cartcomm);
                //MPI_Allreduce(MPI_IN_PLACE, &w, 1, MPI_DOUBLE, MPI_SUM, \
                proc->cartcomm);

                w /= (NX * NY * NZ);
                e /= w;

                /*
                if(proc->rank == 0)
                {
                        if ((n % 100) == 0 || n == 20)
                        printf("%5d, %0.4e\n", n, e);
                }*/

                n++;
        }

        free(vp);

        if(proc->rank == 0)
        {
                if (e < eps)
                printf("Converged after %d iterations (nx=%d, ny=%d, nz=%d, \
                e=%.2e)\n", n, NX, NY, NZ, e);
                else
                printf("ERROR: Failed to converge\n");
        }

        return (e < eps ? 0 : 1);
}
```