Master in High Performance Computing

Exploring innovative approaches in industrial and academic HPC applications

Supervisor(s):
Dr. Giuseppe Piero Brandino,
Prof. Luca Heltai

Candidate:
Dr. Matteo Barnaba

7th edition
2020–2021

"We are all visitors to this time, this place. We are just passing through. Our purpose here is to observe, to learn, to grow, to love ... and then we return home."

Australian Aboriginal saying

Acknowledgments

This thesis is dedicated to my family and to my girlfriend Laura, thank you for all your love and support.

I thank my friends for their support and the big eXact lab family, especially Francesco and Pino, for the tremendous amount of things they taught me.

Contents

Chapter 1

Introduction

In this thesis are presented some of the results obtained as contribution to two different projects.
The first one is the REDSEA project, an EU founded project whose goal is to develop exascale network interconnect, described in more detail in section 1.1.
The contribution to this project is, in turn, two-fold:
First are presented the results related to the collection and analysis of network traces for the applications contributed by eXact lab.
This activity is aimed at suggesting critical requirement for the network characteristics, in a co-design approach.
The traces were collected for the LAMMPS molecular dynamics package and the results are presented and discussed in chapter 2.

The other contribution to the REDSEA project is the porting of applications to the project specific architecture.
In particular, a parallel version of the Self Organizing Maps (SOM) algorithm has been developed.
The goal was to develop a massively-parallel implementation of this algorithm, based on MPI and optimized for the RED-SEA architecture.
The implemented code, available on github, has also full support for OpenSHMEM parallelization, with the main goal being the ability to run on GSAS, a Partioned Global Address Space (PGAS) environment developed by FORTH.
The results related to these activities, along with a comparison between the two implementations, are presented and discussed in chapter 3.

The activities related to the second project presented in this thesis are discussed in chapter 4.
The main task was to extend the SOFA framework (via a plugin) to allow harnessing the ray-tracing cores of modern GPUs to perform collision detection.
The SOFA framework is a powerful framework used for real-time FEM based simulations, with applications, among many others, in surgery and engineering.
The developed collision detection pipeline employes the NVIDIA OptiX ray-tracing Engine that enabled us to exploit dedicated, and very powerful, hardware to perform collision detection, a very relevant task for many scientific and industrial applications.

1.1 The REDSEA project

Network interconnects play an enabling role in HPC systems. and This is particularly true for coming Exascale systems that will rely on higher node counts and increased use of parallelism and communication. Moreover, next-generation HPC and data-driven systems will be powered by heterogeneous computing devices, including low-power ARM and RISC-V processors, high-end CPUs, vector acceleration units and GPUs suitable for massive single-instruction multiple-data (SIMD) workloads, as well as FPGA and ASIC designs tailored for extremely power-efficient custom codes.

These compute units will be surrounded by distributed, heterogeneous (often deep) memory hierarchies, including high-bandwidth memories and fast devices offering microsecond-level access time. At the same time, modern data-parallel processing units such as GPUs and vector accelerators can crunch data at amazing rates (tens of TFLOPS).
In this landscape, the network may well become the next big bottleneck, similar to memory in single node systems.

RED-SEA will build upon the European interconnect BXI (BullSequana eXascale Interconnect), together with standard and mature technology (Ethernet) and previous EU-funded initiatives to provide a competitive and efficient network solution for the exascale era and beyond.
This involves developing the key IPs and the software environment that will deliver:

- scalability, while maintaining an acceptable total cost of ownership and power efficiency;
- virtualization and security, to allow various applications to efficiently and safely share an HPC system;
- Quality-of-service and congestion management to make it possible to share the platform among users and applications with different demands;
- reliability at scale, because fault tolerance is a key concern in a system with a very large number of components;
- support of high-bandwidth low-latency HPC Ethernet, as HPC systems increasingly need to interact securely with the outside world, including

public clouds, edge servers or third party HPC systems;

- support of heterogeneous programming model and runtime to facilitate the convergence of HPC and HPDA;
- support for low-power processors and accelerators.

The main objectives of the REDSEA project are to:

- Enable the design of a new generation of high performance network interconnect leveraging existing European technology (BXI, Exanest) able to power the future EU Exascale systems
- Explore new innovative solutions end-to-end network services – from programming models to reliability, security, low latency, and new processors
- Develop the ecosystem and create a broader community of users and developers leveraging open standard and compatible API to develop innovative re-usable libraries and Fabrics management solutions

Chapter 2

MPI tracing

2.1 LAMMPS

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a classic molecular dynamic engine with a focus on material modeling. It is used widely in several branches of science:

- solid state physics
- computational chemistry
- biophysics
- many others

It is also well known for its ease of compiling and running on several different computer architectures (from laptop to large cluster). The fact that it is used to simulate dynamics for atomic (atomic gases), meso (large molecules such as proteins) and continuum scale (metals), makes it a perfect co-design reference tool.

The parallelization of LAMMPS is based on the $3D$ domain decomposition paradigm, in which every MPI rank is in charge of a subdomain containing a subset of all the atoms. Moreover, the assignment of atoms to a specific domain decomposition needs to be update once in a while, to keep into account

possible movement of atoms between domains. This means that the communication pattern contains point-to-point communications to transfer information to nearby domains, plus some collective operations (mostly Broadcast and AllReduce) to update the domains population.
Moreover, when long range interactions such as Coulomb interaction are present, the calculation of forces requires Ewald summation method (such as particle-particle particle mesh, PPPM) which in turn require distributed Fourier Transform.
Distributed Fourier Transform require communication of data between all domains that shares either the X, Y or Z index in the $3D$ domain grid.

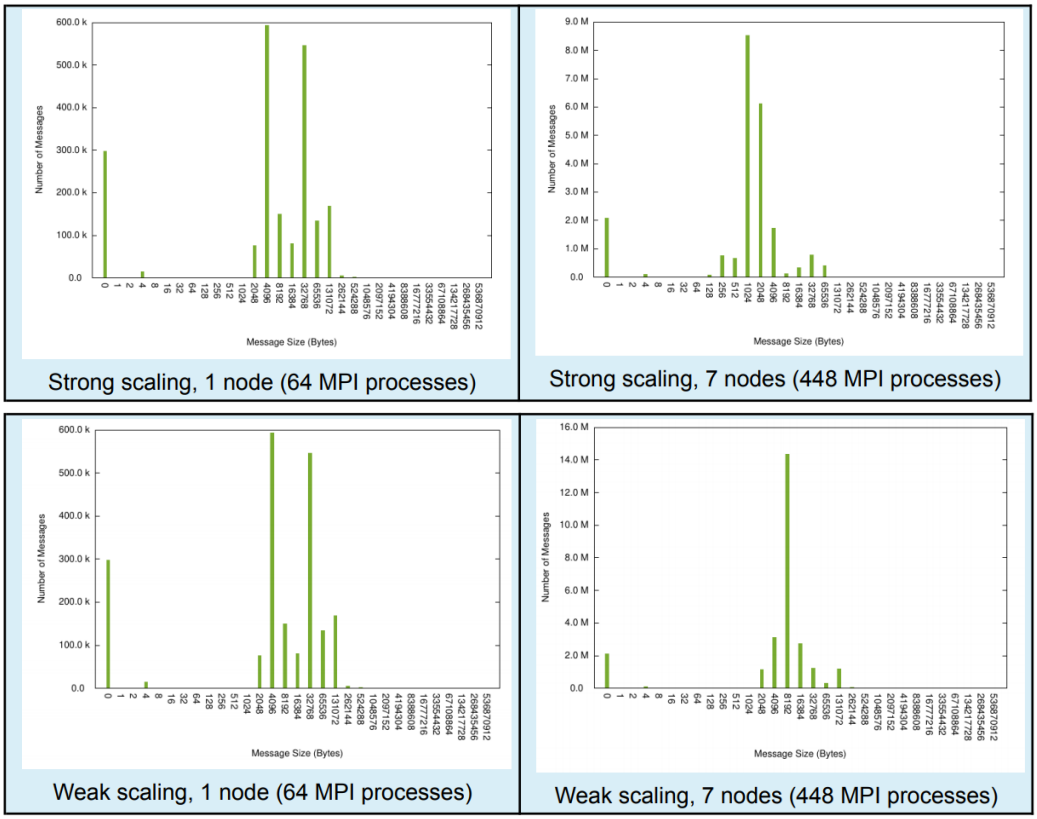

Figure 2.1: Point-to-Point message distribution sizes for the rhodopsin benchmark. Strong scaling setup: 2.048.000 atoms. Weak scaling setup: 32000 atoms per MPI process.

2.2 Results

The runs were performed in the Dibona cluster using one MPI process per core and a single OpenMP thread per MPI rank.
The analysis has been performed for the rhodopsin benchmark from LAMMPS. This example considers an all-atom rhodopsin protein in solvated lipid bilayer with CHARMM force field, long-range Coulombic interaction via PPPM (particle-particle particle mesh) and SHAKE constraints, containing 32000 atoms.
This example has been chosen due to its computational complexity and network load compared to other molecular dynamics simulation, because of the need of calculating $3D$ Fourier transform for the PPPM method. In order to have a significant computational load, the system has been replicated a

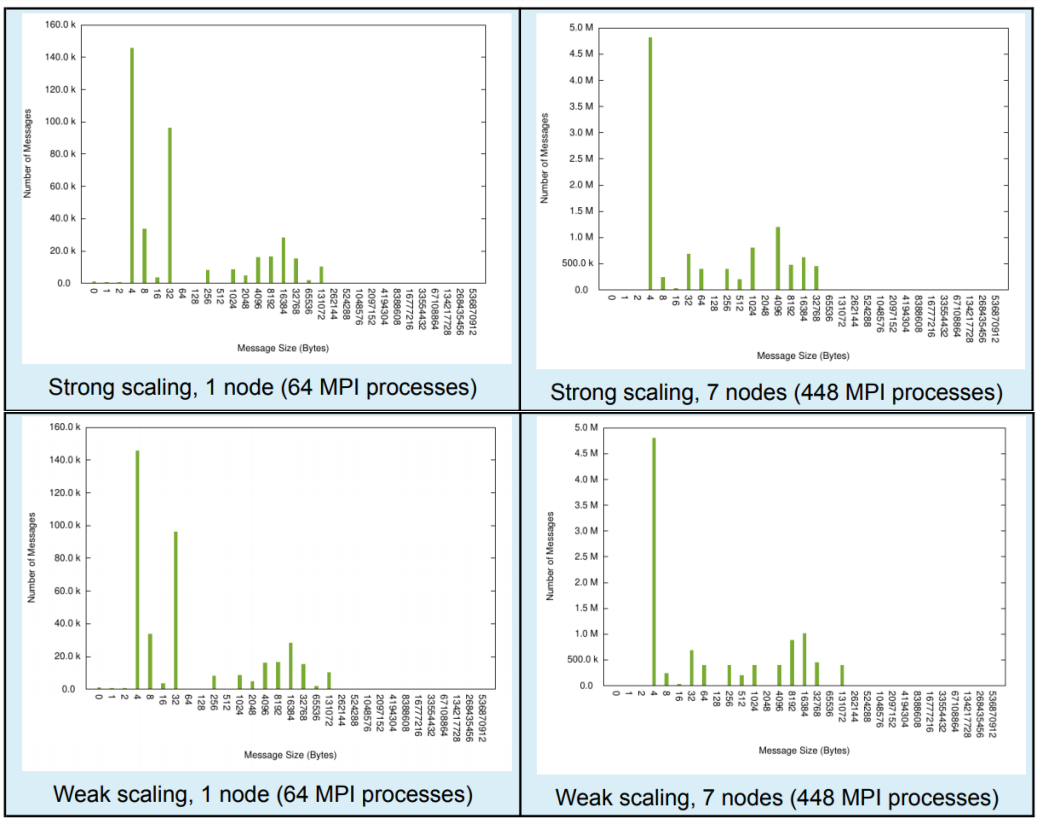

Figure 2.2: Collective message distribution sizes for the rhodopsin benchmark. Strong scaling setup: 2.048.000 atoms. Weak scaling setup: 32000 atoms per MPI process.

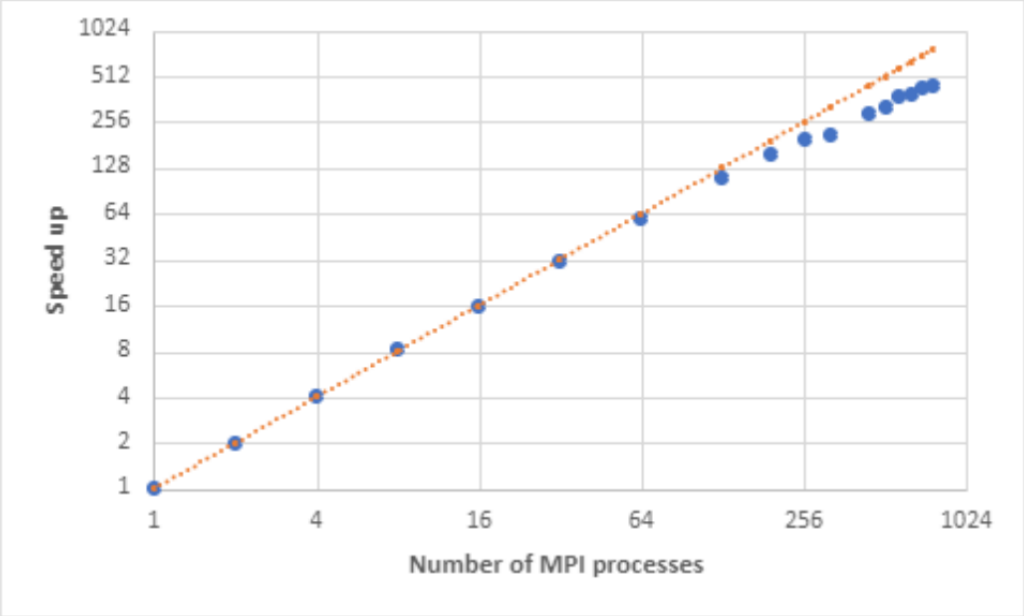

Figure 2.3: Parallel speedup of the rhodopsin benchmark in the strong scaling setup: 2.048.000 atoms.

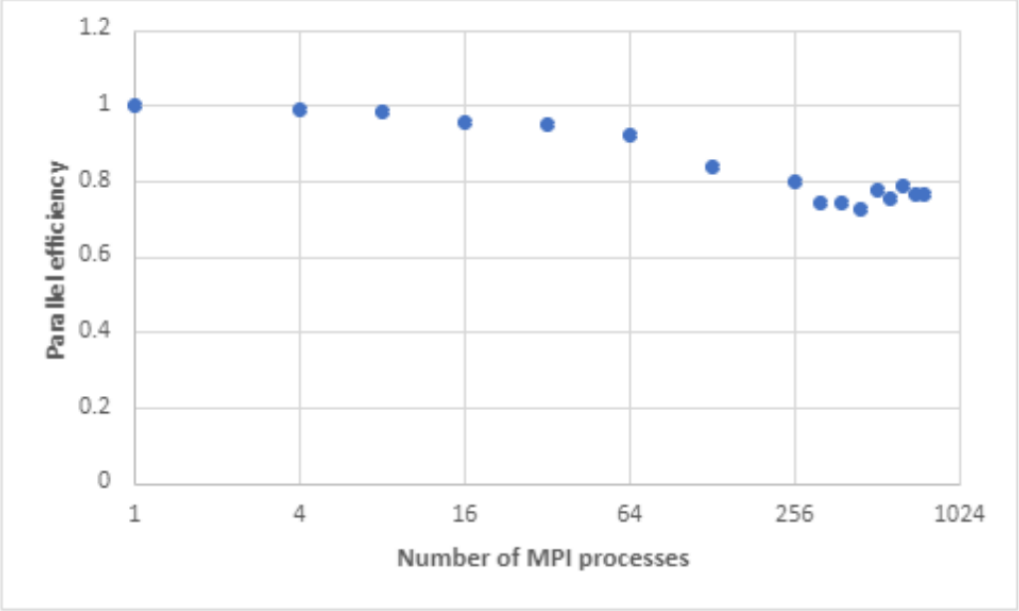

Figure 2.4: Parallel Efficiency in the weak scaling setup: 32.000 atoms per MPI process.

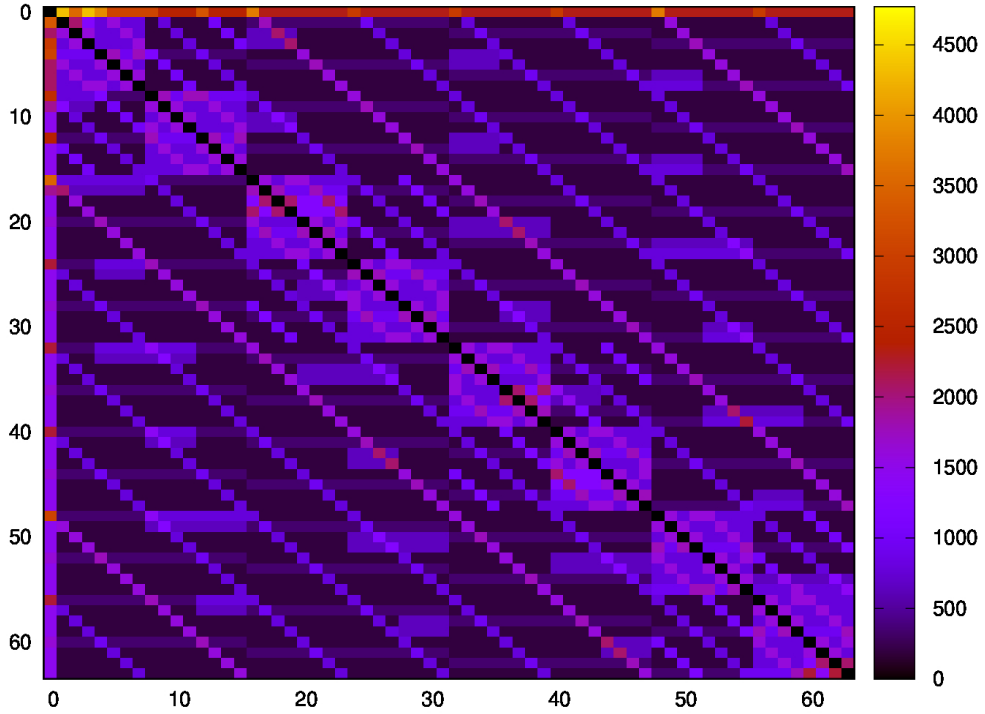

Figure 2.5: Global call matrix. The x-axis shows the rank of the sending node, while the y-axis of the receiving node.

number of times with the following setups:

- For strong scaling runs, the system has been replicated four times in all spatial direction, for a grand total of 2.048.000 atoms.
- For weak scaling runs, each MPI rank was in charge of dealing with the fundamental cell of 32000 atoms.

The runs were performed in the Dibona cluster using one MPI process per core and a single OpenMP thread per MPI rank.
Figure 2.1 reports the comparison between two different configurations of the message size distribution for MPI point-to-point communication, with 1 node (64 MPI processes) and 7 nodes (448 MPI processes) respectively, both for strong and weak scaling setups. The x-axis shows the message sizes in logarithmic scale, while the y-axis shows the number of messages.

In the strong scaling setup, we can see a definite trend both in the message sizes and message number.
Regarding the message sizes, in the strong scaling settings we should expect a near linear decrease in the message sizes, since for both the communications of particle data and Fourier transform, the amount of data to be transferred is proportional to the volume of the domain, and increasing the number of MPI tasks corresponds to linearly decrease the volume of the domains (more precisely, particle data behaves linearly plus some second order corrections in the ration between the skin size and the domain size).
The plots shown are roughly compatible with this interpretation. The change in shape of the distribution is due to the aforementioned higher order corrections and communication optimizations.

Regarding message numbers, the scaling behaves differently for the particle data and Fourier transform:
increasing the number of domains increases the number of messages for particle data linearly, since each domain has to communicate only with neighboring ones.
For the Fourier transform instead, each domain has to communicate with all the domains on at the same x, y, and z. For a cubic setting with N domains per side, this amounts to $3n^2(n-1) \approx n^4$ point-to-point communication per Fourier transform call.
This implies that passing from K to αK MPI processes, the size of the domains is scaled down by a factor $\alpha^{1/3}$, which implies that the expected

number of messages increases by a factor $\alpha^{4/3}$.
This insight is compatible with the results in figure 2.1.

The conclusion is then that in the strong scaling setup, the sizes of messages are typically small and scales down linearly, while the number of messages scales up superlinerly.

In the weak scaling setup instead, the range in message sizes does not change significantly, since the size of the domains is fixed, while the number of messages is expected to show the same behavior as in the strong scaling case, since this quantity depends on the number of domains and not from their sizes.
The single peak in the 7 nodes setting in weak scaling is interpreted as the merge of the two other peaks, and the scaling analysis above confirms that intuition.

Collective MPI communications in LAMMPS are dominated by Broadcast and AllReduce. Message size distribution are reported in figure 2.2 for both strong and weak scaling. As in the point-to-point case, the message sizes scales down in the strong scaling setups while they do not change in the weak scaling setups. Message number instead scales linearly in both cases as expected.

In conclusion, the message sizes for complex molecular dynamics simulations involve typically small messages and in a number that scales superlinearly in the number of MPI processes. This is reflected in the execution time.
Figure 2.3 and 2.4 show the parallel speed up and parallel efficiency in strong and weak scaling respectively.
Figure 2.3 shows two different kinds of trend, before and after 64 MPI processes. It is important to remember that each node on the Dibona cluster is equipped with 64 cores, so moving above 64 MPI processes corresponds to the onset of inter-node communications. The different trends are interpreted as the difficulty of the network in handling the large number of small messages generated by the simulation.
This change however could also be due to the decreasing size of the messages, so it could be as well a natural behavior of the parallelization algorithm. However, a similar shift in trends is seen also in the weak scaling regime, where the message sizes do not change.
This strengthens the intuition that the network has difficulty in handling the large number of small messages.

Figure 2.5 is instead a graphical representation of the traffic patterns as a matrix plot for a 64 MPI processes run. The x-axis shows the rank of the sending node, while the y-axis of the receiving node. The plot includes both collective and point-to-point communications, while the color indicates the number of messages. The plot shows rather complex patterns and in general a high connectivity between nodes. This is mostly due to the Fourier transform, since, as discussed before, each domain has to separately communicate with all the domains at the same x, y and z. In a 64 MPI processes run the domains are arranged in a 4x4x4 configuration, that explains the communication pattern spacing of multiples of 4.

2.3 Conclusions

The results presented in the previous section show that the network should be able to handle well small messages and in a number that scales superlinearly with the number of MPI processes.
In fact, for a strong scaling setup, the sizes of the messages are typically small and scales down linearly with the number of MPI processes.

On the other hand, in a weak scaling setup, message sizes do not vary significantly with the number of MPI processes since the size of the domains does not vary much with the number of MPI processes
As already mentioned, the number of messages scales up superlinearly with the number of MPI processes for both the strong and weak scaling setup.

The FFT, needed for systems characterized by long-range interactions, is the main responsible for the observed scaling of the message size and therefore the major bottleneck for the parallel scaling of LAMMPS as a whole.
Preliminary analysis is currently being performed with particular attention to DEEP-SEA results about GROMACS FFT bottleneck.

Chapter 3

Self-organizing maps

3.1 SOM

Self-organizing maps (SOM) (e.g., Kohonen, 1985, 1995; Lawrence et al., 1999; Wittek et al., 2017; Silva and Marques, 2007) are artificial neural networks that are used in the context of unsupervised machine learning.
SOM techniques have been successfully applied in a number of disciplines including speech recognition, image classification and document clustering.
In SOM networks high-dimensional input data is projected onto a low-dimensional (typically two-dimensional) array. This nonlinear projection produces a low-dimensional (typically two dimensional) "feature map" that can be used to detect and analyze features/properties of the input space.

Supervised learning is used to train the network against input data with known outputs.
In contrast, the SOM technique is typically applied to data in which specific classes or outcomes are not known a priori, and hence training is done in unsupervised mode.
In this case, the SOM can be used to understand the structure of the input data, and in particular, to identify "clusters" of input records that have similar characteristics in the high-dimensional input space.
After the training is performed, new input records (with the same dimension as the training vectors) can be analyzed (and assigned to clusters) using the neural weights computed during training.

The SOM technique is able to produce a structured ordering of the input vectors, i.e., similar vectors in the input space are mapped to neighboring neural nodes in the trained feature map.
This "self-organization" is particularly useful in clustering analysis since it provides additional insight into relationships between the identified clusters.

3.2 Implementation

The in-house implementation of the parallel SOM (Self-Organizing map) algorithm developed by eXact lab has been written in C++17 and C11 and is available on github.
The software has been developed as a library (usable as a standalone library) and an executable that uses this library and provides an easy-to-use interface for it.
The implemented training algorithm could be described as an adaptation/enhancement of the "data-partitioned batch SOM algorithm" (Lawrence et al., 1999).

The original algorithm can be summarized as follows.

- At the beginning of the training process the input data is distributed (equally) among the running parallel processes/tasks.
- For each epoch/iteration of the training process then each task processes only the portion of the input data that has been assigned to it in the previous step and accumulates locally its contributions to the global state.
- After each task has completed its local accumulation, collective communications are used to combine the local contributions and place the results in all tasks.
- At this point the global state of the training process can be updated and next epoch/iteration of the training process can begin.
- The training process ends when the maximum number of epochs/iterations is reached or when the training process does not modify the global internal state more than a given minimum threshold.

The developed implementation of the parallel SOM algorithm introduces various important enhancements to the algorithm just described.
Firstly, the number of updates of the global state of the training process can be set at will.
In the original algorithm the global state is updated only once per epoch (i.e., after all the input data is presented to the network).

This design, in our opinion, leads to a level of granularity that is too coarse and thus we decided to modify the algorithm so that the batchsize (i.e., the number of records to be presented to the network before updating the global state) can be set at will.

Our implementation of the algorithm has permitted to achieve a very good balance between performance (speed and parallel scalability of the algorithm) and the quality and granularity of the results.

In order to achieve the aforementioned goals also the starting step of the training processes, in which input data is distributed and assigned to the parallel processes/tasks, has been carefully designed to attain the maximum performance during the training procedure.

The developed implementation of parallel SOM algorithm has full support for both MPI and OpenSHMEM parallelization.
The main goal of the OpenSHMEM implementation is the ability to run on the GSAS, a Partioned Global Address Space (PGAS) environment developed by FORTH.
Moreover, a serial implementation (which does not need any parallelization library) has been developed with the main goal of serving as a reference for the results produced by the two parallel implementations.
In fact the results produced by the 2 parallel implementations are perfectly identical to the ones produced by the serial implementation.

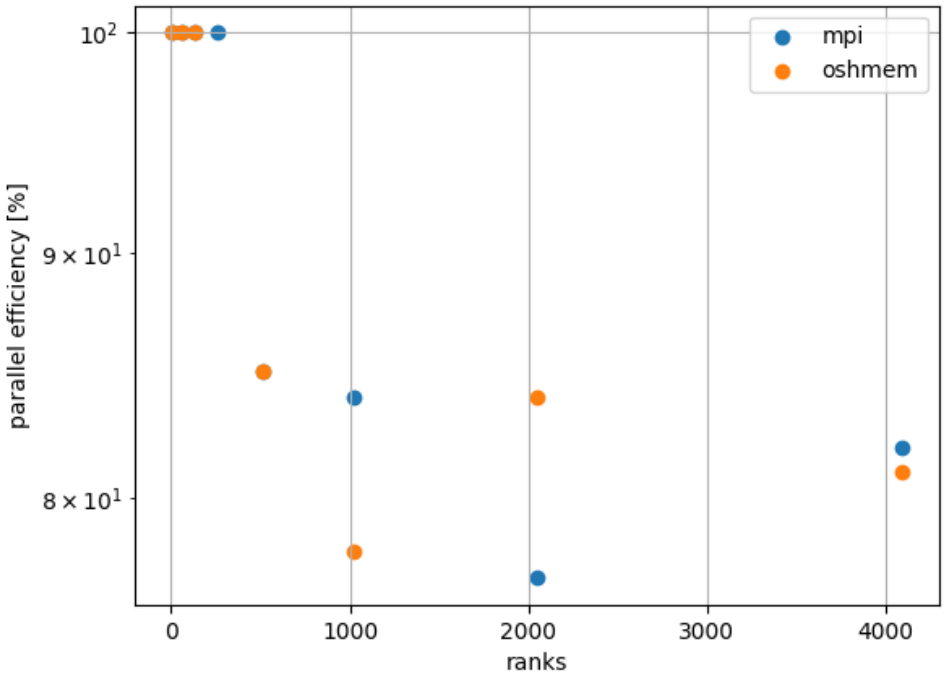

Figure 3.1: Parallel efficiency measured on the TGCC KNL testbed for a dataset comprised of $1e8$ records, with 2 features each, for a number of ranks ranging from 16 to 4096 and 1 batch per epoch. Comparison between the OpenMPI and the OpenSHMEM implementation.

3.3 Results

The results collected show that the developed parallel SOM implementation is characterized by a very good performance, both in terms of speed and parallel scalability.
In figure 3.1 is depicted the parallel efficiency for a dataset comprised of $1e8$ records, with 2 features each, for a number of ranks ranging from 16 to 4096 and for 50 batches per epoch.
Figure 3.1 offers also an interesting comparison between the OpenMPI and the OpenSHMEM implementation.
The presented results have been collected on the TGCC KNL testbed, but similar results have also been collected on other machines.

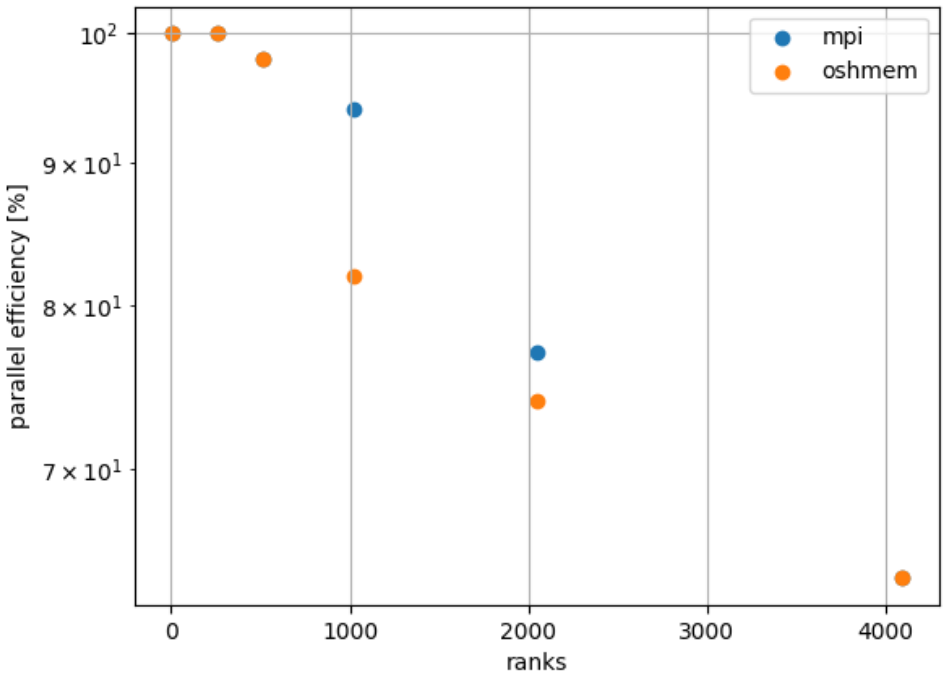

Figure 3.2: Parallel efficiency measured on the TGCC KNL testbed for a dataset comprised of 1e8 records, with 2 features each, for a number of ranks ranging from 16 to 4096 and 50 batches per epoch. Comparison between the OpenMPI and the OpenSHMEM implementation.

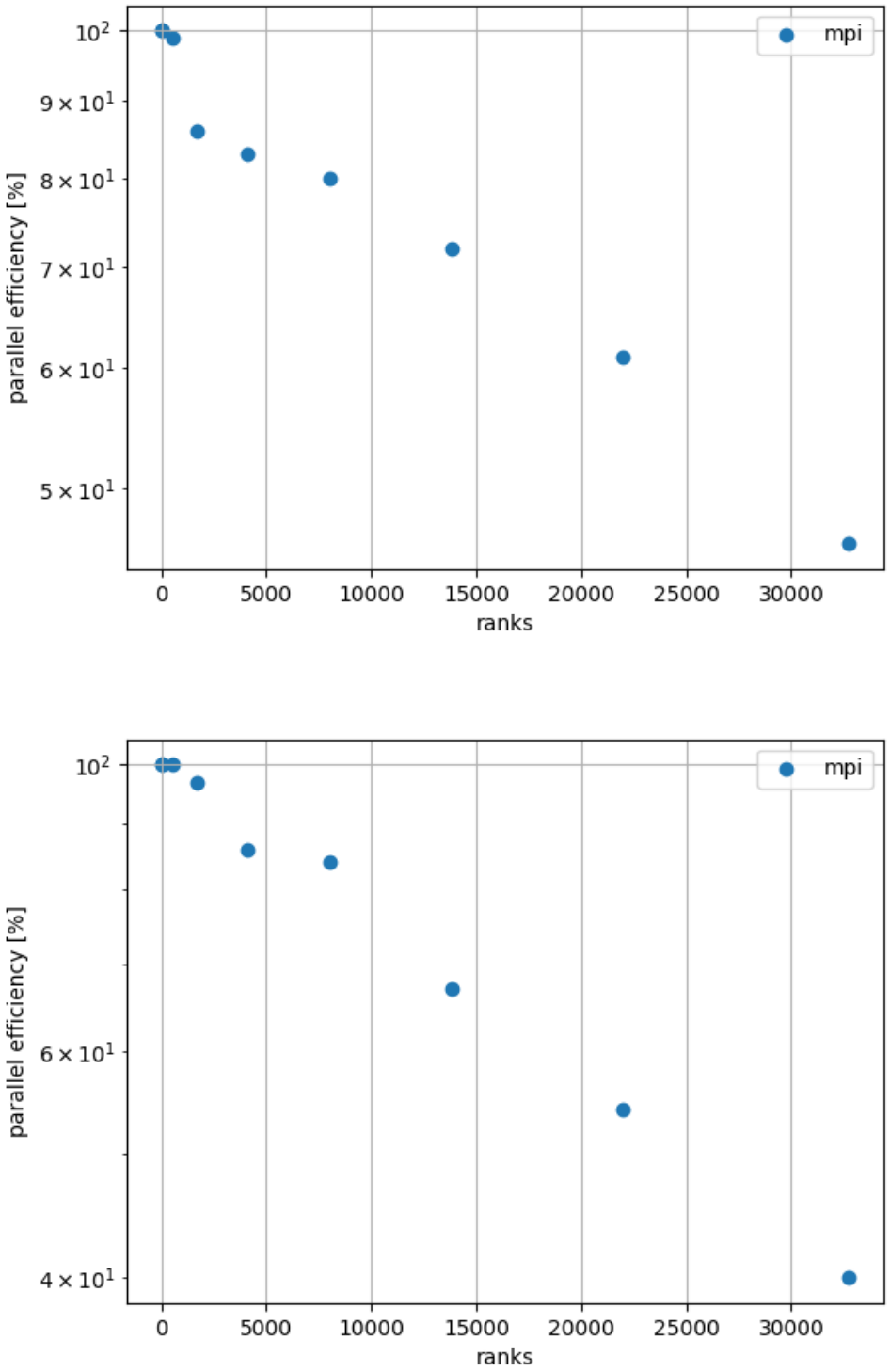

Figure 3.3: Parallel efficiency measured on the TGCC KNL testbed for a dataset comprised of $1e9$ records, with 2.5 features each, for a number of ranks ranging from 64 to 32768. 10 (top) and 100 (bottom) batches per epoch.

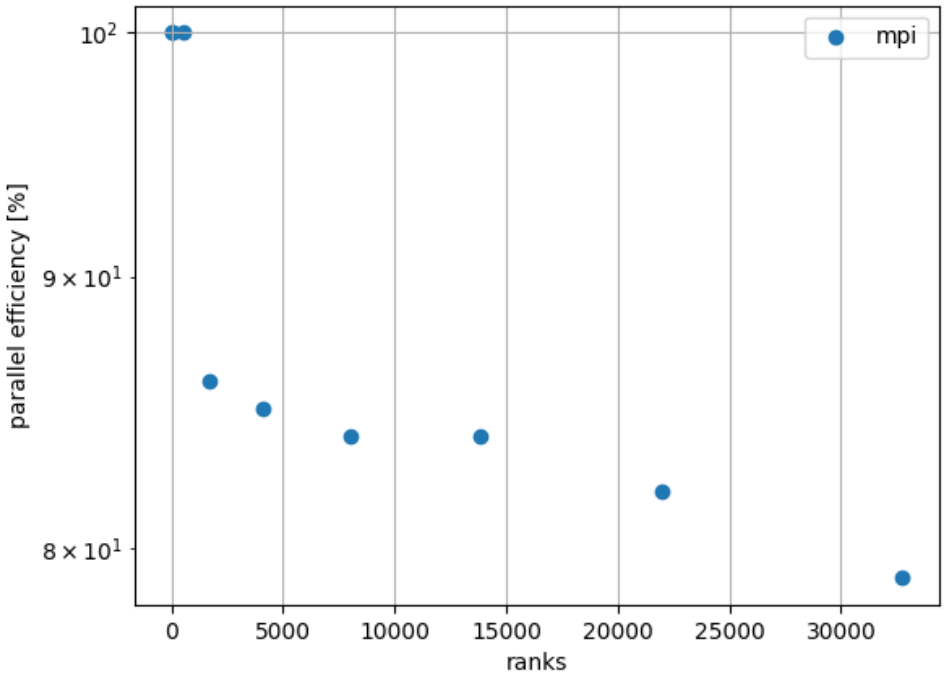

Figure 3.4: Parallel efficiency measured on the TGCC KNL testbed for a dataset comprised of $1e9$ records, with 2 features each, for a number of ranks ranging from 64 to 32768. 1 batch per epoch.

The presented results, considering the big number of ranks, show in general a good scaling capabilities for the implemented code.
The TGCC KNL machine is equipped with nodes with 64 cores (where the sudden drop in performance is found in the presented figures), so the results presented in figure 3.1 and figure 3.2 where obtained engaging 64 nodes, while the results presented in figure 3.3 and figure 3.4 (up to 32768 ranks) where obtained using 512 nodes.

In figure 3.2 are depicted the results for a run that is almost identical to the one presented in figure 3.1, the only difference is the number of batches per epoch.
Going from 1 batch per epoch (all the records are presented to the lattice before a global update of the map) to 50 batches per epoch, we see a noticeable decrease in performance. Of course this is due to the increased communication needed between all MPI ranks, 50 times greater than in the previous

case (figure 3.1).

This deterioration of the performance is not due entirely to the implementation of the algorithm, but also to the size of the dataset.
In fact, increasing the size of the dataset (the amount of computation to perform) the deterioration in performance kicks-in later and later, thus for higher and higher numbers of MPI ranks.
This can be seen in figure 3.3, in which are presented the parallel efficiencies related to a dataset 10 times bigger ($1e9$ records) than the one considered for figures 3.1 and 3.2 and for a number of MPI ranks ranging from 64 to 32768.
As the number of batches per epoch increases from 10 to 100, the point in which the deterioration in performance kicks-in is associated to smaller number of MPI ranks.
However this number is much bigger than the one associated to the run from figures 3.1 and 3.2.
Thus it could be said that the implemented algorithm is characterized by a good scalability as long as the ratio between the time spent in communication and the time spent in computation is low, as it has to be expected.
Finally figure 3.4 depicts the parallel efficiency considering the same dataset as figure 3.3 but only 1 batch per epoch, showing a quite good behavior on the entire range of MPI ranks (64 - 32768).
Again, notice how the deterioration in performance kicks-in much later that in figure 3.1, associated to a dataset 10 time smaller.

Unfortunately it was not possible to collect the results related to the OpenSHMEM implementation of the algorithm for the cases depicted in figures 3.3 and 3.4 because OpenSHMEM is no longer available on TGCC KNL after the last system maintenance.

Chapter 4

Collision detection with NVIDIA OptiX

4.1 NVIDIA OptiX

NVIDIA OptiX is a CUDA-centric API that is invoked by a CUDA-based application. The API is designed to be stateless, multi-threaded and asynchronous, providing explicit control over performance-sensitive operations like memory management and shader compilation.

It supports a lightweight representation for scenes that can represent instancing, vertex- and transform-based motion blur, with

- built-in triangles;
- built-in swept curves;
- built-in spheres;
- user-defined primitives;

NVIDIA OptiX implements a single-ray programming model with ray generation, any-hit, closest-hit, miss and intersection programs.
The ray tracing pipeline provided by NVIDIA OptiX is implemented by 8 types of programs:

1. Ray generation: the entry point into the ray tracing pipeline, invoked by the system in parallel for each pixel, sample, or other user-defined work assignment.
2. Intersection: implements a ray-primitive intersection test, invoked during traversal.
3. Any-hit: called when a traced ray finds a new, potentially closest, intersection point, such as for shadow computation.
4. Closest-hit: called when a traced ray finds the closest intersection point, such as for material shading.
5. Miss: called when a traced ray misses all scene geometry.
6. Exception: exception handler, invoked for conditions such as stack overflow and other errors.

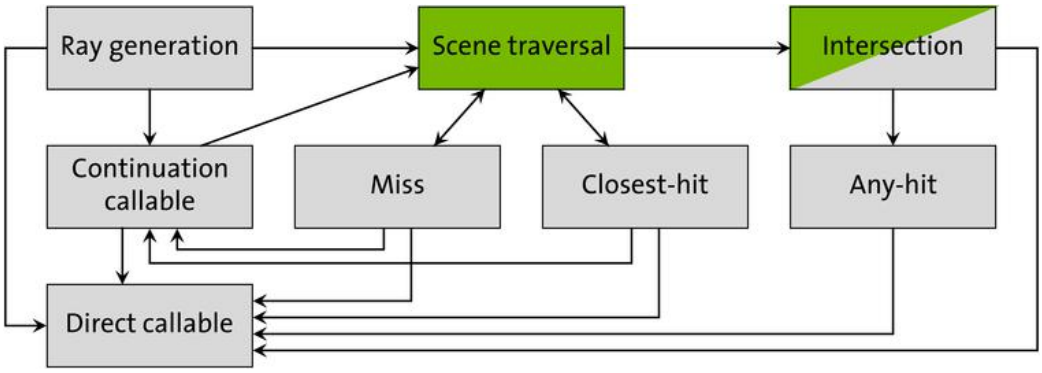

Figure 4.1: Relationship between the 8 types of OptiX programs.

7. Direct callables: similar to a regular CUDA function call, direct callables are called immediately.
8. Continuation callables: unlike direct callables, continuation callables are executed by the scheduler.

The ray-tracing pipeline is based on the interconnected calling structure of these 8 programs and their relationship to the search through the geometric data in the scene, called a traversal. In figure 4.1 is presented a diagram of the relationship between these types of programs.

In OptiX a functional ray-tracing system is implemented by combining 4 components as described in the following steps:

1. Create one or more **acceleration structures** over one or many geometry meshes and instances of these meshes in the scene.
2. Create a **pipeline of programs** that contains all programs that will be invoked during a ray tracing launch.
3. Create a **shader binding table** that includes references to these programs and their parameters and choose a data layout that matches the implicit shader binding table record selection of the instances and geometries in the acceleration structures.
4. Launch a **device-side kernel** that will invoke a ray generation program with a multitude of threads calling **optixTrace** to begin traversal and the execution of the other programs.

4.2 SOFA

SOFA is an open source framework targeted at interactive physics simulation, with an emphasis on medical simulation and robotics. It is mainly intended for the research community to help foster newer algorithms, but can also be used as an efficient prototyping tool. SOFA's advanced software architecture allows:

- creation of complex and evolving simulations by combining new algorithms with existing algorithms
- modification of key parameters of the simulation such as deformable behavior, surface representation, solvers, constraints, collision algorithm, etc.
- synthesis of complex models from simpler ones using a graph description
- efficient simulation of the dynamics of interacting objects using abstract equation solvers
- comparison of various algorithms and mathematical models

SOFA is often presented as a standalone software (runSofa) and a simulation tool, but the project is most importantly a bundle of libraries and thus can be used/integrated in any project.

SOFA provides a plugin system allowing the coupling of additional codes to add functionalities.

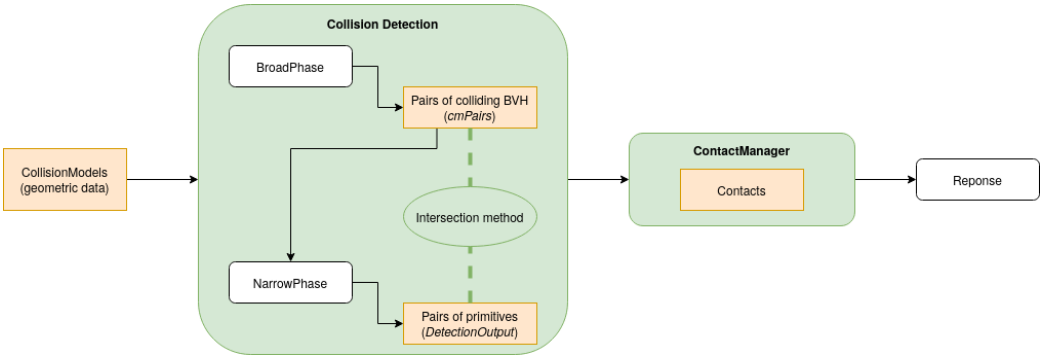

Figure 4.2: Phases of the collision-detection pipeline.

4.2.1 Collision detection

In all SOFA simulations the collision phase is done separately from the physics simulation, and usually before the call to the solvers.
Collision detection is split in several phases, each implemented in a different component and each phase is scheduled by the collision pipeline.

The collision pipeline follows three steps:

- reset of the collisions from the previous step (if any)
- collision detection
- collision response

The first step of the list is quite self-explanatory.
The collision detection step aims at determining if two (or several) objects collide. In SOFA, the collision detection takes as input the collision models (geometric data) and returns pairs of geometric primitives as output, along with the associated contact points. This contact information is then passed to the contact manager, which creates contact interactions of various types based on customizable rules.

Given Number of objects moving objects in a virtual environment, testing all objects pairs tend to perform Complexity of pairwise checks pairwise checks. When Asymptotic complexity complexity, the collision detection is usually divided into two successive steps:

- a broad phase
- a narrow phase

The first step (broad-phase) aims at quickly and efficiently removing objects pairs that are not in collision. The broad phase uses a set of root collision models in order to compute potentially colliding pairs. It can for instance rely on the bounding boxes of each object with a collision model, thus efficiently checking whether boxes collide or not.
This step does not state if pairs of objects collide, but it detects if they potentially collide.

The narrow phase, on the other hand, is the phase in which intersections (contacts) are actually detected. It relies on collision models to detect a contact, for example, the most used models are primitives: point, line, triangle, sphere, cube, cylinder or oriented bounding boxes (OBB).
All collision detection methods rely also on intersection methods during the broad and/or narrow phase in order to assess if the models do collide, i.e., given 2 collision elements, intersection methods test if an intersection is possible.

The output of the collision-detection phase are pairs of geometric primitives with the corresponding collision points. The collision information is saved in a vector of structures named DetectionOutput, a generic description of a contact point and which holds the following fields:

- elem: a pair of colliding elements
- id: unique id of the contact for the given pair of collision models. Used to filter redundant contacts (only the contact with the smallest distance is kept);
- point: contact points on the surface of each model;
- normal: normal of the contact, pointing outward from the first model;
- value: signed distance (negative if objects are interpenetrating);
- deltaT: estimated of time of contact;

Figure 4.2 depicts the aforementioned collision-detection phases and their relationship with the other parts of the collision pipeline.

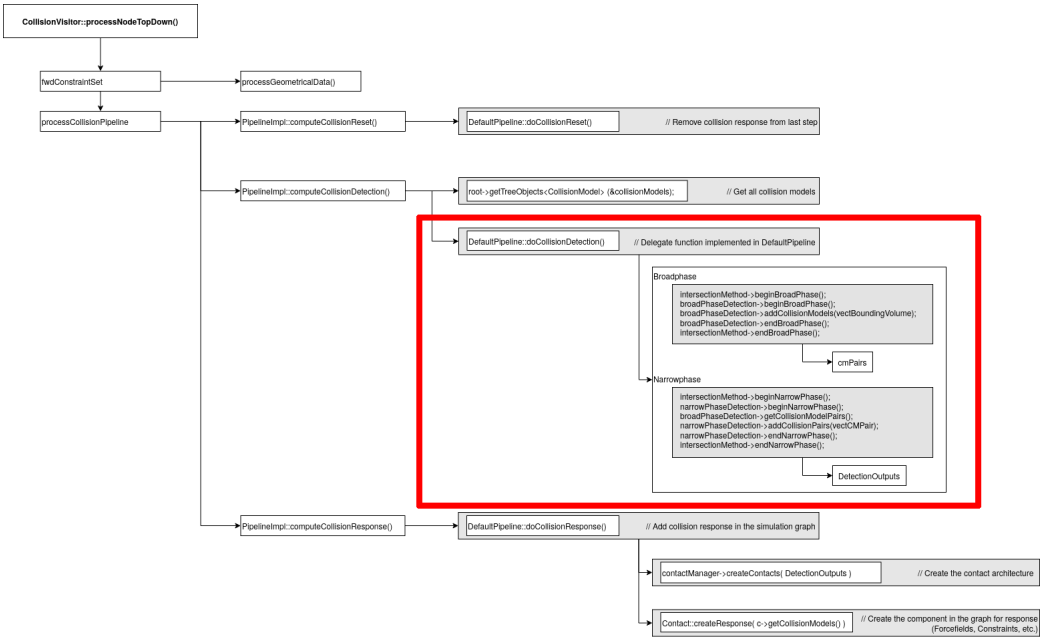

Figure 4.3: Steps of the collision pipeline. The steps in the red rectangle are covered and replaced by the developed pipeline.

4.3 Implementation of the plugin

After the functioning of both OptiX and SOFA were briefly explained, it is possible to illustrate the implementation of the developed collision-detection pipeline that employs the OptiX API.
The core idea can be graphically expressed via figure 4.3: replace all the steps, of the SOFA collision pipeline, inside the red rectangle with the offloading of the models to the GPU and take advantage of its ray-tracing capability to identify which elements are about to collide.
As already stated in the introductory part of this chapter, the ray-tracing is completely hardware-accelerated when using triangles (among few others) as geometric primitives, thus associated with very good performance, as it will be seen in the results section.

In order to harness the ray-tracing capability of the GPU for collision detection, it is necessary to first translate a physical model to a purely geometric one that can be expressed via simple primitives. This is because OptiX is designed for graphic and does not know anything about complex models and their behavior. It only knows about a simple scene described via simple geo-

metric primitives (triangles, curves, spheres) and about a ray, described via a starting point and a direction (both in a 3D space).
It is important to note that both OptiX and SOFA use the concept on a "scene" but unfortunately they refer to two completely unrelated concepts.
For this reason it is necessary to "explode" the SOFA collision models into the triangles of which their surface is comprised and keep track of which triangle came from which collision model for the later creation of the contacts. Of course it is not necessary to take into account the triangles that are internal to a mesh because they are protected by the external elements and cannot be collided before the external ones.

If the collisions models are treated as a set of triangles, then a collision between two models takes place if there are two triangles (one per model) that collide.
In turn two triangles collide if at least one of vertices of one triangle collides with the surface of the other.
From these considerations comes the fact that to check whether (and where) two complex models collide it is sufficient to check whether (and where) any on the vertices of one model collides any the triangles (their surface) of the other model.
Finally it comes the core idea of the implemented algorithm: for each triangle in each model send (with a direction normal to the triangle's surface) a ray from its 3 vertices and check whether (and where) a triangle is hit by the ray. This is the perfect fit for a ray-tracing engine such as OptiX and if a hit is identified and the distance between the vertex, from which the ray originated, and the hit surface is less than a certain threshold, than a collision is identified.
Note that this way each pair of models are tested for collisions twice: the first time for a model are considered the vertices and the directions for the rays, while the second time its triangles are considered. This way all possible collision are accounted for.

```
unsigned triangleIndex;
unsigned short vertexIndex, componentIndex;
for (triangleIndex = 0; triangleIndex < numTriangles; ++triangleIndex)
{
    const auto& modelTriangle = modelTriangles[ triangleIndex ];
    Triangle& triangle = append( triangles );

    for (vertexIndex = 0; vertexIndex < 3; ++vertexIndex)
    {
        const auto& modelVertex = x[ modelTriangle[ vertexIndex ] ];
        Vertex& vertex = triangle.v( vertexIndex );

        for (componentIndex = 0; componentIndex < 3; ++componentIndex)
            vertex.c( componentIndex ) = modelVertex[ componentIndex ];

        invertMap.add(
            modelIndex,
            triangleIndex,
            vertexIndex
        );
    }

    const auto& modelNormal = modelNormals[ triangleIndex ];
    Normal& normal = append( normals );

    for (componentIndex = 0; componentIndex < 3; ++componentIndex)
        normal.c( componentIndex ) = modelNormal[ componentIndex ];
}
```

Listing 4.1: Collection of the triangles for the offloading.

In practice the first step of the algorithm is to collect the triangles comprising the surface of the collision models and store (copy) them into a continuous chunk of memory (necessary for CudaMemcpy).
This step is necessary because (at the present time) the vertices of the triangles are not stored in a contiguous fashion and in a format that fits the one requested by OptiX.
In this context a triangles (in 3D) is simply represented by 9 floats (3 per vertex), listing 4.1 presents the code to extract the triangles from a given collision model.
An additional data structure, called invertMap, is populated during this step in order to speedup the interpretation of the results produced by OptiX.
The triangles (represented by the 3 vertices) are stored in a contiguous fashion, regardless of the model from which they originated. It is thus necessary an auxiliary data structure to easily (and fast) reconstruct, in case a collision is detected for a vertex by OptiX, from which model and triangle that vertex originated.
OptiX in turn will simply return, for each vertex, the index of the hit triangle, if any.
This simple form of the interface to OptiX has a two-fold benefit:

- The GPU-side code is plain simple (no unnecessary looping on all possible pairs of collision-models) and easy to run in parallel since each

vertex is associated to a thread (from which it will perform the ray-tracing on all the triangles). See the code executing the ray-tracing ("raygen" callback) in listing 4.2, the results are stored in the array named "vertexHitTriangle".
Note that also the normal to each triangle is passed to OptiX (instead of having the GPU evaluate it for each triangle), because they are available on the SOFA-side and so that each CUDA thread really has to deal with and care about only one vertex.

- Since each vertex can intersect also the triangles originated from the same model the vertex originated, it is possible to account effortlessly for auto-collisions (parts of a collision model colliding with other pairs of the same model), which can be a very important phenomenon for extended and deformable models.

```
const OptixTraversableHandle traversableHandle{
    $pipelineParametersGet( traversableHandle )
};
const float3* vertices{ $pipelineParametersGet( vertices ) };
const float3* normals{ $pipelineParametersGet( normals ) };
assert( traversableHandle != 0 );
assert( vertices != nullptr );
assert( normals != nullptr );


const unsigned vertexIndex{ optixGetLaunchIndex().x };
const unsigned triangleIndex{ vertexIndex / 3 };
assert( triangleIndex < $pipelineParametersGet( numTriangles ) );


const float3& start{ vertices[ vertexIndex ] };
const float3& direction{
    float3Scale( normals[ triangleIndex ], -1.0f )
};


const float minDistance{ 1.0e-4 };
const float maxDistance{ $pipelineParametersGet( contactDistance ) };
const float rayTime{ 0 }; // rayTime -- used for motion blur

unsigned hitTriagleIndex{ 0 };
optixTrace(
    traversableHandle,
    start,
    direction,
    minDistance,   // minDistance
    maxDistance,   // maxDistance
    rayTime,
    OptixVisibilityMask( 255 ), // always visible
    OPTIX_RAY_FLAG_DISABLE_ANYHIT,
    0,             // SBT offset
    1,             // SBT stride
    0,             // miss SBT index
    hitTriagleIndex
);
assert( hitTriagleIndex <= $pipelineParametersGet( numTriangles ) );


unsigned* vertexHitTriangle{ $pipelineParametersGet( vertexHitTriangle ) };
assert( vertexHitTriangle != nullptr );

vertexHitTriangle[ vertexIndex ] = hitTriagleIndex;
```

Listing 4.2: Raygen callback starting the ray-tracing on the GPU.

In listing 4.2 notice also how the “any-hit” callback can be disabled (leaving active only the “closest-hit” and the “miss-hit” callbacks) via the “OPTIX_RAY_FLAG_DISABLE_ANYHIT” flag, resulting in an improvement in performance. It is possible to set this flag because, looking for collisions, we are just interested in the first object hit by the ray (the closest one) and none of the others.
Listing 4.3 presents the simple code (thanks to the OptiX device-side interface) executed by OptiX when the closest hit (for a particular ray) occurs, the “closest-hit” callback.

```
__global__
void
__closesthit__function()
{
    optixSetPayload_0(
        optixGetPrimitiveIndex() + 1
    );
}
```

Listing 4.3: Closest-hit gpu-side callback.

4.4 Results

The developed collision detection pipeline employing NVIDIA OptiX has been compared to the SOFA pipeline
"sofa::component::collision::detection::algorithm::DefaultPipeline" and the comparison has been performed in terms of the time spent for the collision detection step (doCollisionResponse() method).
Two SOFA pipelines, CPU and CUDA (GPU), were considered. The later version has been generated changing the "template" field for the relevant objects in the scene (see using CUDA).

The time spent for the collision detection step has been measured for different "sizes" of a simple base scene (size 1).
The base scene was composed of a pair of cubes: a moving one and a still one, acting as an obstacle for the first one (see figure 4.4).
The other sizes of the scene were obtained by increasing the number of moving cubes and increasing the size of the still cube (the obstacle), that becomes a rectangle, to accommodate the increased size of whole scene.
A simple Python3 script was created to generate those scenes, for different sizes and implementations: OptiX, SOFA CPU and SOFA CUDA.
Figure 4.5 presents the scene for 10 moving cubes that will collide (falling on it) with the bigger still rectangle.
This simple setup was chosen because it is possible and simple to reason about the number of collision that are expected and to check whether all the expected collisions are actually detected.
In fact, being the scene composed by n moving (falling) cubes that fall on 1 large still rectangle, the number of expected collisions is simply $4n$, where the factor 4 comes from the fact that, for each moving cube, 4 vertices face (downward) the still rectangle acting as an obstacle.
The distance between the moving cubes was chosen so that no collisions between them are possible, thus to have no spurious collisions tampering the results with noise.

Not all the steps of the simulations are characterized by $4n$ collisions detected. For a few beginning steps the number of collisions is less than $4n$ because none collision is detected when the moving cubes and the obstacle are at a distance greater that the one that triggers the collision detection "alarmDistance". As the distance decreases, the number of collisions increases until it reaches $4n$.
For the comparison between the developed collision detection pipeline and

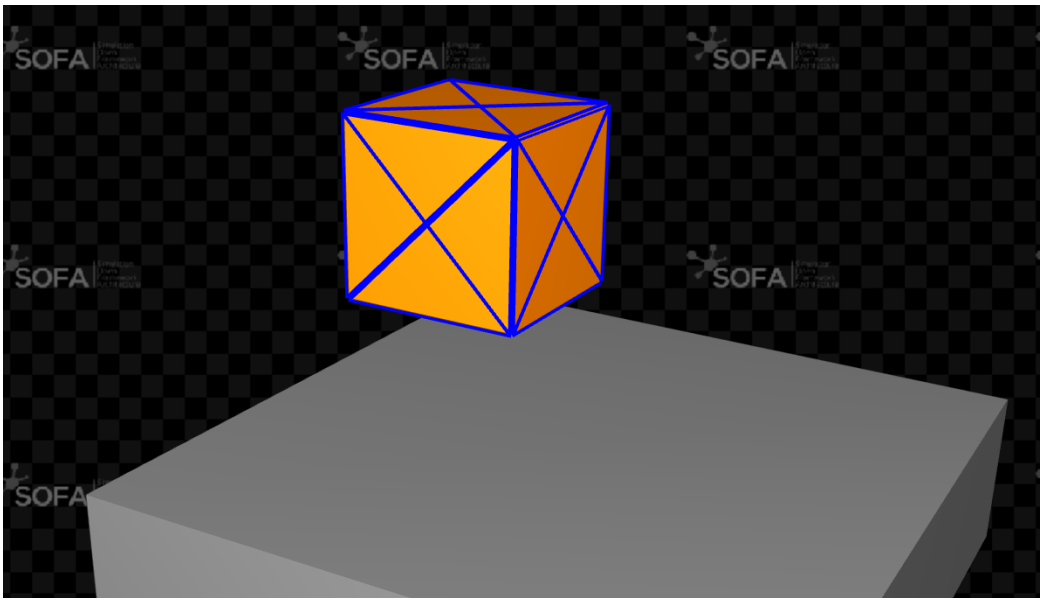

Figure 4.4: SOFA scene with 1 moving cube.

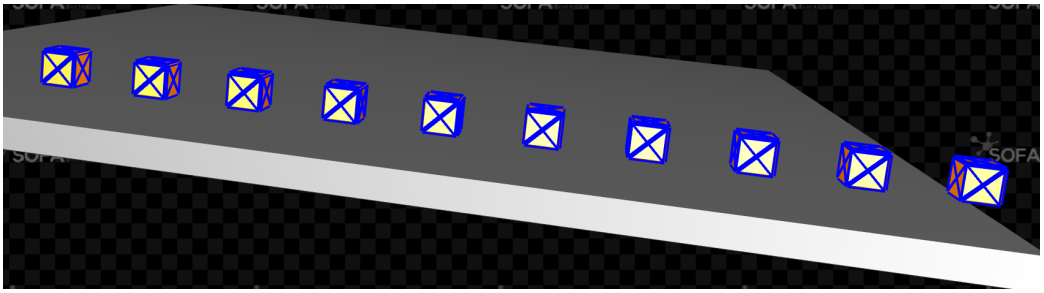

Figure 4.5: SOFA scene with 10 moving cube.

cubes	optix-seconds	cpu-seconds	cpu-speedup	cuda-seconds	cuda-speedup
1.000e+00	1.641e-04	1.122e-05	6.907e-02	2.515e-06	1.547e-02
2.000e+00	1.906e-04	2.016e-05	1.064e-01	4.044e-06	2.134e-02
1.000e+01	2.429e-04	1.101e-04	4.559e-01	2.122e-05	8.785e-02
1.000e+02	4.354e-04	3.109e-03	7.203e+00	7.011e-04	1.623e+00
5.000e+02	9.523e-04	5.826e-02	6.490e+01	1.458e-02	1.623e+01
1.000e+03	1.620e-03	2.302e-01	1.536e+02	5.623e-02	3.755e+01
1.500e+03	2.274e-03	5.199e-01	2.524e+02		
2.000e+03	2.884e-03	9.100e-01	3.551e+02		
5.000e+03	6.952e-03	5.640e+00	9.485e+02		
1.000e+04	1.452e-02	2.589e+01	2.141e+03		
2.000e+04	3.239e-02	1.482e+02	5.525e+03		
3.000e+04	4.926e-02	3.970e+02	9.847e+03		

Table 4.1: Time spent for the collision detection (average over the fully-operational steps) by the developed pipeline (optix) and the cpu and cuda SOFA pipelines, along with the related speedups.

the SOFA ones only the simulation steps characterized by $4n$ detected collisions (fully-operational) were considered and the average over these steps was taken.

The results of the comparison are quite positive and encouraging for the employment of NVIDIA OptiX for collision detection.
In fact the developed collision pipeline employing OptiX resulted much faster than the SOFA ones, both for the CPU and GPU version.
Table 4.1 presents a summary of the comparison, in terms of the time spent for the collision detection (average over the fully-operational steps), between the developed pipeline (optix) and the cpu and cuda SOFA pipelines. Also the speedups (with respect to the SOFA cpu and cuda pipelines) are reported in the table.
The same results from table 4.1 are presented graphically in figure 4.6, in terms of the execution time, and in figure 4.7, in terms of the speedup with respect to the cpu and cuda SOFA pipelines.

The results show that the developed pipeline (optix) runs much faster that than both the cpu and cuda SOFA pipelines. With respect to the cpu version the speedup gets to approximately $1e4$ when a scene with $3e4$ moving cubes is considered, which can be thought as a very positive result.
Another positive aspect of the developed collision pipeline is that its computational time seems to grow much slower (with respect with the two SOFA pipelines) with the number of detected collisions.
This behavior can be seen in figure 4.6 and explains the increase of the of speedup with the number of collisions (number of moving cubes) depicted in

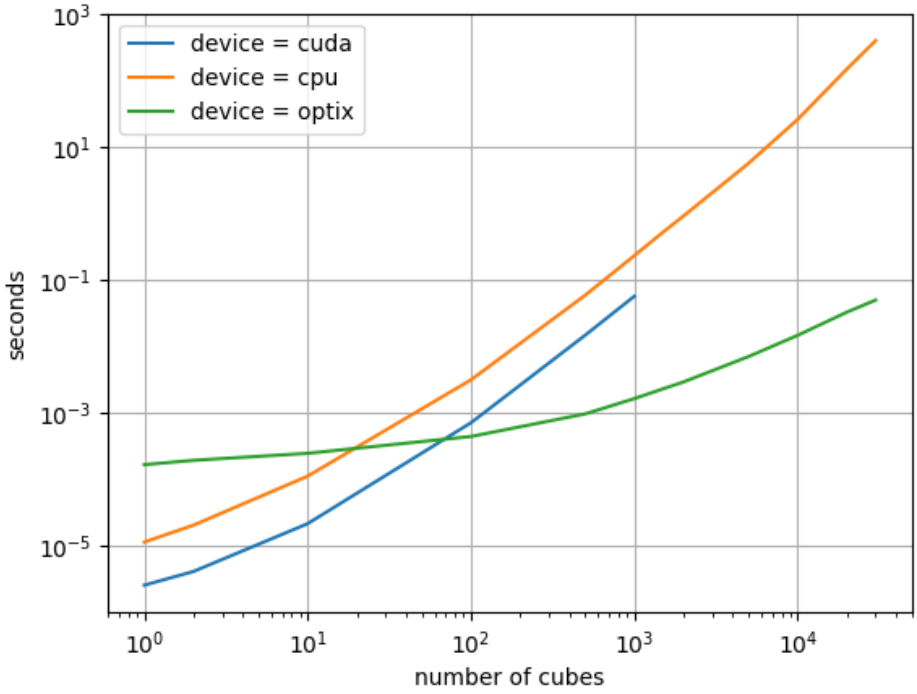

Figure 4.6: Time spent for the collision detection (average over the fully-operational steps) by the developed pipeline (optix) and the cpu and cuda SOFA pipelines. The data is presented in table 4.1.

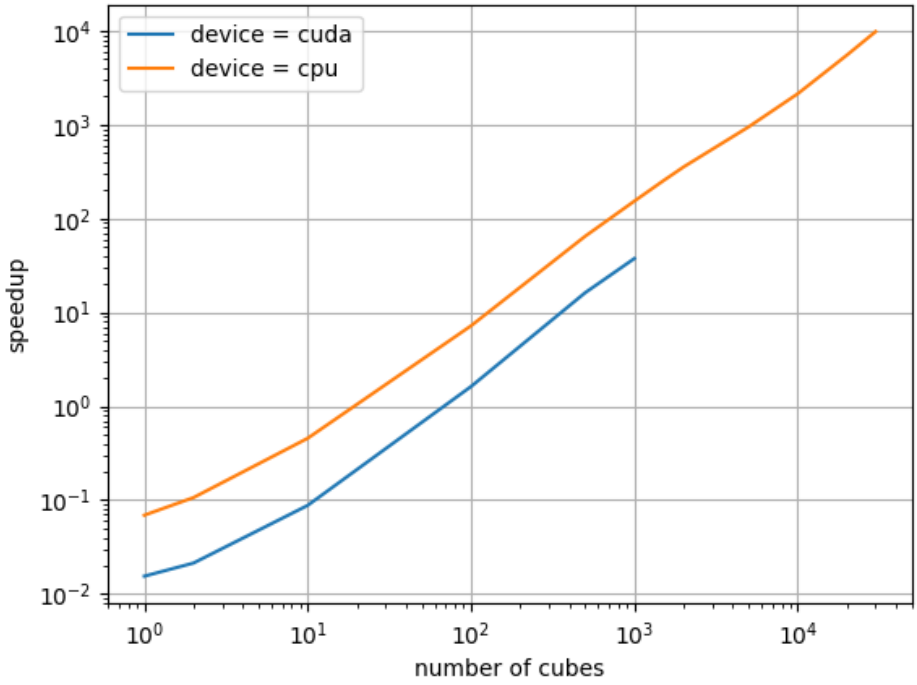

Figure 4.7: Speedup of the developed pipeline (optix) with respect to the SOFA cpu pipeline and the SOFA cuda pipeline. The data is presented in table 4.1.

figure 4.7.

Table 4.1 and figures 4.6 and 4.7 do not present the results for the SOFA cuda pipeline for scenes with more than $1e3$ moving cubes because this pipeline was not able to run above this threshold. This pipeline crashed because it was not possible to allocate all the necessary memory on the GPU ("cudaMalloc error out of memory").
At $1e3$ moving cubes the speedup with respect to the cuda SOFA pipeline is approximately 37, which can be thought as a good result. It is also believed that the comparison between the developed pipeline and this SOFA pipeline is not entirely fair for the developed pipeline. In fact, while the SOFA cuda pipeline already has its data on the GPU when collision detection starts, the developed pipeline has to gather this data and send it to GPU before collision detection can start.
It is thus believed that the speedup could be higher than the reported one if the developed pipeline had its data already on the GPU (like the cuda pipeline) at the beginning of collision detection. This could be perhaps achieved by re-implementing the developed pipeline to store its data (in a OptiX-friendly fashion) directly on GPU, thus removing the translation phase between SOFA and OptiX (described earlier in this chapter) from the pipeline.

As it can seen from the results, the only areas for which the developed pipeline seems to be slower then the two SOFA pipelines are small scenes, i.e., characterized by a small number of collisions.
This fact is easy to explain considering that, for the developed pipeline, a phase of translation between SOFA and OptiX (described earlier in this chapter) is necessary and the fact that sending to and retrieving data from the GPU can be expensive. Both the operations are not worth the effort when the amount of work for the GPU is little.

Bibliography

[1] Kohonen, T. and Saarinen, J. *Self-Organized Formation of Colour Maps in a Model Cortex* Perception, 14(6), 711–719. (1985). https://doi.org/10.1068/p140711

[2] Kohonen, T. *Self-Organizing Maps* Springer Series in Information Sciences (Vol. 30). Berlin, Heidelberg. (1995).

[3] Lawrence, R., Almasi, G. and Rushmeier, H. *A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems* Data Mining and Knowledge Discovery 3, 171–195 (1999). https://doi.org/10.1023/A:1009817804059

[4] Wittek, P. *Training emergent self-organizing maps on sparse data with Somoclu* http://peterwittek.com/training-emergent-self-organizing-maps-withsomoclu.html (2017).

[5] Silva, Bruno, and Nuno Marques *A hybrid parallel SOM algorithm for large maps in data-mining* New Trends in Artificial Intelligence (2007).

[6] François Faure, Christian Duriez, Hervé Delingette and Jérémie Allard *SOFA: A Multi-Model Framework for Interactive Physical Simulation* Soft Tissue Biomechanical Modeling for Computer Assisted Surgery, 2012

[7] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. *OptiX: a general purpose ray tracing engine* ACM Trans. Graph. 29, 4, Article 66 (July 2010), 13 pages, 2010. https://doi.org/10.1145/1778765.1778803

[8] Tomas Nikodym *Ray Tracing Algorithm For Interactive Applications* Czech Technical University, FEE, (June 2016).