Possibilities for Parallelizing Simplicial Complexes Simulation

Dušan Cvijetić, Nenad Korolija, and Marko Vojinović

***Abstract*—This manuscript presents potentials for parallelizing simulation of simplicial complexes. The implementation of most important fields and methods of classes for storing simplicial complexes and k-simplices is followed by wrapper classes for simplicial complexes and k-simplices respectively. Infrastructure for communication between Message Passing Interface (MPI) processes along with helper functions is explained further in the manuscript. Once multiple data are prepared to be sent from each MPI process to other MPI processes, sending and receiving is performed in the background. Because of the stall introduced by using MPI directives, the amount of data to be transmitted is maximized by processing multiple operations over simplicial complexes in parallel. This requires the method for locking simplicial complexes and k-simplices by the owner MPI process until all the requests are processed. Locking mechanism and supporting simplicial complex class actions regarding locking is not in the scope of this manuscript.**

***Index Terms*—Simplicial complex; k-simplex; triangulation; manifold; MPI; parallelization.**

I. Introduction

In modern theoretical physics, a lot of problems are too complicated for study using analytical methods, and one needs to resort to numerical techniques. Among those problems, an especially important class deals with evaluation of functions over simplicial complexes. A simplicial complex [1] is a piecewise-linear approximation of a smooth spacetime manifold [2] and is typically 4-dimensional or higher. Functions over a simplicial complex represent physical fields on spacetime, and one commonly employs path integral evaluations of such structures to extract expectation values of observables. For example, in Lattice Quantum Chromo-dynamics, one employs such numerical techniques to predict the theoretical values for the masses of elementary particles called hadrons [3]. Also, in Causal Dynamical Triangulations approach to quantum gravity [4,5], one uses these techniques to evaluate spectral dimension of spacetime, and study various properties of phase space of triangulated manifolds. Finally, in the Regge Quantum Gravity approach [6,7,8] one can study the entanglement properties of matter fields and gravity described by the Hartle-Hawking wavefunction [9,10], again using the techniques of numerical evaluation of path integrals over simplicial complexes.

It goes without saying that all such calculations are exceptionally expensive in computation time. Typically, one develops custom-made code, heavily optimized to solve precisely one specific problem, and executes it over months-long periods on hardware dedicated for high performance computing (HPC), usually clusters with thousands of work nodes. Such enormous calculational efforts are usually unavoidable due to the nature of the problems that need to be solved.

Nevertheless, at least for one class of such problems, it may be possible to construct a more general algorithm and structures which would provide a common basis for solving an all-encompassing class of problems using the same underlying software, while intrinsically exploiting the parallelization possibilities of the code itself and the distributed nature of the underlying hardware. Our aim is to develop such a generic software library, which could be used to solve a whole host of physics problems in the same way and optimize it for parallelized HPC environments. In this work we present the first steps towards the construction of such a library. This approach of developing common code for a whole class of problems has not been attempted so far because research teams are usually concentrated on solving only one specific problem and opt to construct custom code for that problem. However, in our opinion, a generic software library, which would provide support for a whole class of problems simultaneously, would open new avenues for numerical research, since one could use the same code to study new, yet unexplored problems as well as old well-known ones.

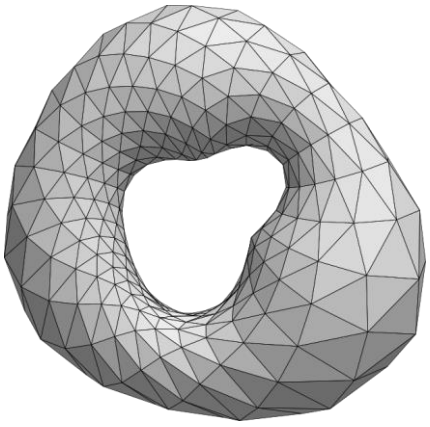

Fig. 1. Simplicial complex of a torus (source: Wikipedia).

The fundamental structure which lies at the core of the whole numerical method is the notion of a *simplicial complex*. A simplicial complex is a combinatorial structure which is easiest to understand as a generic lattice-like mesh,

Dušan Cvijetić is a student of the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: dusancvijetic2000 @ gmail.com).

Nenad Korolija is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: nenadko @ etf.bg.ac.rs).

Marko Vojinović is with the Institute of Physics, University of Belgrade, Pregrevica 118, 11080 Pregrevica, Serbia (e-mail: vmarko @ ipb.ac.rs).

 ISBN 978-86-7466-930-3

whose cells are called *simplices*, and are connected to each other along their boundaries to form the simplicial complex of a given dimension. The purpose of the whole structure is to approximate the smooth spacetime manifold with a discrete structure which is more convenient for numerical methods.

The most elementary simplex is a simplex of level zero, often called *0-simplex* or *vertex* – it is just a dimensionless point with no structure. Next is the *1-simplex*, also called an *edge* – it is a one-dimensional line with two vertices at its boundaries. At level two we have the *2-simplex* or *triangle*, whose boundary are three edges and their vertices. The *3-simplex*, also known as the *tetrahedron*, has the boundary made of four triangles and their edges and vertices. The procedure of constructing simplices can be done for arbitrary dimension, giving rise to the notion of a *k-simplex*, whose level (i.e. natural dimension of space in which it is defined) is equal to any positive integer k. The most commonly used example is the *4-simplex*, also called *pentachoron* – a 4-dimensional figure whose boundary consists of 5 tetrahedra, 10 triangles, 10 edges and 5 vertices. In most applications in physics, the spacetime manifold is considered to be 4-dimensional, and it is cut into a lattice-like structure made of 4-simplices, which are glued together along their boundary tetrahedra. The resulting structure is a simplicial complex of dimension 4. Fig. 1 depicts an intuitive example of a 2-dimensional simplicial complex of a torus.

Given a simplicial complex, one typically wants to introduce functions that are evaluated on it. These are commonly called *colors* and are assigned via their values to each k-simplex within in the complex. In other words, some colors live on vertices, some on edges, some on triangles, and so on. The colors are a natural discretization of the notion of a *field* over a manifold. For example, just like electric and magnetic fields have a value at each point of a smooth spacetime, analogously the colors have values at each k-simplex in the simplicial complex.

Depending on the type of the problem at hand, algorithms that are used to evaluate required quantities on a simplicial complex can vary in complexity, from conceptually simple Monte Carlo integration techniques, to vastly complicated traversal and ray-tracing algorithms, to various methods for solving functional partial differential equations. Due to the variability of the complexity of all these algorithms, dictated by the nature of the problem at hand, it is helpful to develop the underlying software simulator to exploit the parallelization avenues that are intrinsic to the simplicial complexes and k-simplices themselves, so that the simulator can exploit parallel hardware environments even for algorithms that are themselves hard to parallelize. This helps the code developer with overall optimization and application to HPC hardware architectures. In what follows, we shall demonstrate a set of possible approaches to these intrinsic parallelization techniques.

II. N-Dimensional Simplicial Complexes

This section describes data structures used in the simulator of simplicial complexes from the point of view of their suitability for parallelizing the simulator execution. Data demanding structures are of main interest for optimizing the communication between processing units. Along with those, data that describes the structure and needs to be updated on multiple processing units will be described in detail. Further, the amount of data that needs to be exchanged and the frequency of expected changes will be compared to the pyramid, where top elements demand less memory, but require more often communication.

The parallelization is simulated using the MPI framework. The simulator is implemented in C++, and, as a result, the parallelization framework is built on top of the simulator. As improving the simulator of simplicial complexes is an ongoing process, the possibility for accelerating the computation is simulated based on the requirements.

Simplicial complexes are formed out of k-simplices at various levels. Simplicial complexes at level zero represent vertices. The structure of each vertex is stored in *KSimplex* class. Simplicial complexes at level one represent edges. Each edge consists of two vertices. As it is the case with vertices, information about edges are also kept in a *KSimplex* class. However, while vertices can be independent of other vertices, representing separate simplicial complexes, each edge must have at least two vertices defined as neighbors. Neighbor of an k-simplex is defined also as a k-simplex that the first k-simplex relies on. Neighboring relation is symmetrical. Therefore, if two vertices are neighbors of an edge, edge is also the neighbor of both vertices. Further, edges can form a triangle. By analogy, neighbors of triangle are three edges, but also the triangle is neighbor of these edges. The neighboring relation spans more than one level up or down. The triangle has also three vertices as neighbors and the opposite.

Simplicial complex representing a triangle consists of a k-simplex representing a triangle along with all neighbors of the triangle. Simplicial complex class is used for storing information about simplicial complexes. As it has elements field that is a pointer to pointer of k-simplices, it is also used for keeping neighbors of each k-simplex.

III. Parallelizing Simplicial Complexes Simulation

Parallelizing operations over simplicial complexes is implemented by splitting the structure over multiple MPI processes. First, we can consider a single simplicial complex system, as the most general approach. If no screen coordinates for k-simplices are assigned, we can artificially assign this type of color, so that we can present k-simplices in 2D space. Further, we can imagine multiple planes, where each plane is responsible for keeping k-simplices of one dimension. This way, we can consider n-dimensional simplicial complex as a pyramid that we observe from the bird's eye view. Now we could have a bottom-up approach, where k-simplices of dimension zero are divided onto MPI processes based on their screen coordinates. Going up, each MPI process would store higher dimensional k-simplices that have those that are one level below as their neighbors. When a k-simplex has neighbors on one level below that belong to multiple MPI processes, this k-simplex gets copied to all MPI processes involved. Finally, all MPI processes would keep the highest-level k-simplex. In the case of multiple simplicial complexes, they could be split over MPI processes based on the same bottom-up approach.

 ISBN 978-86-7466-930-3

The notion of determining the MPI process where a k-simplex is located is hidden by using wrapper functions, so that the calculation operations are performed as if all k-simplices would have been on the same MPI process, i.e. as if the simulation was executed serially. Each wrapper function can keep either a pointer to the structure, if it exists on the same MPI process, and the ID used for finding the structure on the owner MPI process.

Algorithm 1 describes the most important aspects of simplicial complex classes. First, a basic *SimpComp* class is given, followed by the wrapper class *VirtualSimpComp* used for parallelization.

Algorithm 1: Declaration of simplicial complex classes.

```
class SimpComp{
public:
    SimpComp(int dim);
    SimpComp(string s, int dim);
    ~SimpComp();
    // Creating new KSimplex
    // at level k:
    VirtualKSimplex* create_ksimplex(
        int k);
    void update_owner(int owner);

    string name;
    int D;
    vector< vector<
        VirtualKSimplex *> > elements;
};
class VirtualSimpComp{
public:
    SimpComp *find_simpcomp;

    int id;
    int ownerRank;
    SimpComp *simpComp;
};
```

Algorithm 2 describes the most important aspects of k-simplices classes. A basic *KSimplex* class is followed by the wrapper class *VirtualKSimplex* used for parallelization.

Algorithm 2: Declaration of k-simplex classes.

```
class KSimplex{
public:
    KSimplex();
    KSimplex(int k, int D);
    ~KSimplex();
    bool find_neighbor(
            VirtualKSimplex *k1);
    void add_neighbor(
            VirtualKSimplex *k1);

    int k; // level
    int D; // dimension
    VirtualSimpComp *neighbors;
    vector<Color *> colors;
};
class VirtualKSimplex{
public:
    KSimplex *find_ksimplex();
    int id;
    int ownerRank;
    KSimplex *ksimplex;
};
```

In both algorithms, wrapper functions store a pointer to the base class object, if such exists on a local MPI process. Otherwise, the value is *nullptr*, and the data is searched for on the so called *ownerRank* based on unique identifier called *id*. Owner of this k-simplex can issue multiple requests while it holds a lock.

IV. Infrastructure for Communication Between MPI Processes

The communication between MPI processes is organized as follows. Each MPI process is preparing the data to be sent to other MPI processes. Order of operations prepared for other MPI processes is not important. All requests to other MPI processes for processing are packed in to_rank vector of vectors of unsigned char.

Each type of primitive data is serialized into the array of unsigned characters as it will be explained in the following section. Each prepared byte is pushed to the back of the vector of unsigned characters. Once all the data is prepared, the data is sent to other MPI processes in the background using *MPI_Isend* directive. If a reference to the vector of array of unsigned characters is called *vec*, the pointer to the array is obtained by calling member function *data()* of vector class from standard template library. After issuing all *MPI_Isend* directives, waiting for each of sending to finish is achieved using *MPI_Wait*.

Similarly receiving the data from other MPI processes is implemented in the background using *MPI_Irecv*, followed by *MPI_Wait*, once the data is needed for the processing. The data is received into array of unsigned characters, that is further packed into vector of vectors of unsigned characters called from_rank for simple processing.

V. MPI Supporting Functions

As already mentioned, variables are serialized into the array of unsigned characters using the following syntax:

```
*( (__typeof__ (variable) *) (array + nArray) ) = variable;
nArray += sizeof(variable);
```

Here, *array* is array of unsinged characters where the data stored in the variable is serialized, and *nArray* is the number serialized bytes in the array.

Similarly, a variable is read and prepared into the to_rank using the following syntax:

```
__typeof__ (variable) temp_var = variable; \
int nBytes = sizeof(temp_var); \
for(int iByte = 0; iByte < nBytes; iByte++) \
   to_rank[rankNumber].push_back(
        ((unsigned char *) &temp_var) [iByte] );
```

This can be further optimized, but the optimization is out of the scope of this research.

 ISBN 978-86-7466-930-3

The communication between MPI processes is continued for as long as any MPI process requires further communication with other MPI processes. This is achieved using the following source code, where the MPI process that requires further communication sets variable *to_send* to one:

```
int to_receive = 0; // A rank required communication
MPI_Allreduce(&to_send, &to_receive, 1, MPI_INT,
      MPI_SUM, MPI_COMM_WORLD);
```

After *MPI_Allreduce* is executed, all MPI processes will have the information whether they have to communicate further in *to_receive* variable.

VI. Parallelization Possibilities Using Dataflow Paradigm

This simulator issues the same set of computer architecture instructions repeatedly. As in majority simulator of physical phenomena, the number of instructions is dependent on the precision of the model and is limited by the computing resources and the total simulation time requirement. These conditions are exactly what is required for a program to be suitable for acceleration using the dataflow paradigm [11]. Programming dataflow architectures requires programming skills that are higher than those needed for programming conventional von Neumann architectures. One of the possibilities is to write a program in a VHDL. More suitable solution to most of the programmers would be to exploit the framework that enables writing source code in a Java-like language, which gets automatically translated into the FPGA image [12,13]. Even in this case, the effort needed for programming such architectures is higher [14]. Besides programming dataflow architecture for the simplicial complex simulator, appropriate scheduling scheme is also needed for efficient running of multiple jobs simultaneously [15].

As the number of operations that can be applied to simplicial complexes can lead to several days' simulation time or even more, having in mind the aging and the probability of failure of supercomputing nodes [16], we have decided to write restarts after given number of simulations defined by the user, so that the calculation can continue from the last stored state.

VII. Conclusion

In this work we have presented the basics of the paralellization techniques that can be applied to the structure of a simplicial complex, which underlies a host of research problems in theoretical physics (see also our accompanying paper [17]). These problems tend to be computationally extremely expensive, and the common underlying software that enables parallelization at the level of the basic data structure can possibly go a long way towards optimization of code for numerical study using heavily parallel hardware platforms such as HPC clusters. In particular, the simplicial complex naturally allows for various aspects of parallelization, and we have described the basic classes, corresponding MPI communication infrastructure, supporting functions and the dataflow paradigm employed for the construction.

One should note that our work represents just a first step towards a full working software implementation, and much more effort is needed to properly implement, optimize and test the resulting code in real world environments. All that is the topic for future work. In particular, the data regarding the experimental evaluation, which would compare the proposed parallelization method to ordinary sequential methods still needs to be gathered and analyzed. Nevertheless, this first step is fundamental, and it is conceptually important since it represents a paradigm in which parallelization is implemented dominantly at the level of the simplicial complex as the underlying data structure, rather than at the level of the particular algorithm that aims to solve some particular problem using these data structures.

Finally, we note that our code, once properly developed, may possibly find applications not just in theoretical physics, but also in other disciplines of science, technology and engineering.

Acknowledgment

DC and NK were partially supported by the School of Electrical Engineering, University of Belgrade, Serbia. NK was partially supported by the Institute of Physics Belgrade, contract no. 0801-1264/1. MV was supported by the Science Fund of the Republic of Serbia, grant no. 7745968, "Quantum gravity from higher gauge theory" – QGHG-2021. All authors were partially supported by the Ministry of Education, Science, and Technological Development of the Republic of Serbia.

References

[1] E. H. Spanier, *Algebraic Topology*, New York, USA: Springer Verlag, 1966.

[2] M. W. Hirsch, *Differential Topology*, New York, USA: Springer Verlag, 1976.

[3] S. Durr, Z. Fodor, J. Frison, C. Hoelbling, R. Hoffmann, S. D. Katz, S. Krieg, T. Kurth, L. Lellouch, T. Lippert, K. K. Szabo and G. Vulvert, "Ab-initio Determination of Light Hardon Masses", *Science* **322**, 1224-1227 (2008).

[4] J. Ambjorn, A. Goerlich, J. Jurkiewicz and R. Loll, "Nonperturbative Quantum Gravity", *Phys. Rep.* **519**, 127 (2012).

[5] M. Vojinović, "Causal dynamical triangulations in the spincube model of quantum gravity", *Phys. Rev. D* **94**, 024058 (2016).

[6] T. Radenković and M. Vojinović, "Higher Gauge Theories Based on 3-Groups", *JHEP* **10**, 222 (2019).

[7] A. Miković and M. Vojinović, "Standard Model and 4-Groups", *Europhys. Lett.* **133**, 61001 (2021).

[8] A. Miković and M. Vojinović. "Quantum gravity for piecewise flat spacetimes", *SFIN* **XXXI**, 267 (2018).

[9] N. Paunković and M. Vojinović, "Gauge protected entanglement between gravity and matter", *Class. Quant. Grav.* **35**, 185015 (2018).

[10] J. B. Hartle and S. W. Hawking, "Wave function of the Universe", *Phys. Rev. D* **28**, 2960 (1983).

[11] B. Lee and A. R. Hurson, "Issues in dataflow computing," *Advances in computers*, Elsevier, **37**, 285-333 (1993).

[12] V. Milutinovic, J. Salom, D. Veljovic, N. Korolija, D. Markovic, and L. Petrovic, "Transforming applications from the control flow to the dataflow paradigm," *Dataflow supercomputing essentials*, Springer, Cham, 107-129 (2017).

[13] N. Korolija, J. Popović, M. Cvetanović, and M. Bojović, "Dataflow-based parallelization of control-flow algorithms," *Advances in computers*, Elsevier, **104**, 73-124 (2017).

[14] J. Popovic, D. Bojic, and N. Korolija, "Analysis of task effort estimation accuracy based on use case point size," *IET Software*, **9(6)**, 166-173 (2015).

 ISBN 978-86-7466-930-3

[15] N. Korolija, D. Bojić, A. R. Hurson, and V. Milutinovic, "A runtime job scheduling algorithm for cluster architectures with dataflow accelerators," *Advances in computers*, Elsevier, **126** (2022).

[16] K. Huang, Y. Liu, N. Korolija, J. M. Carulli, and Y. Makris, "Recycled IC detection based on statistical methods," *IEEE transactions on computer-aided design of integrated circuits and systems*, **34(6)**, 947-960 (2015).

[17] D. Cvijetić, N. Korolija and M. Vojinović, "Infrastructure for Simulating n-Dimensional Simplicial Complexes," IcETRAN 2022, Novi Pazar, Republic of Serbia, June 6-9, 2022, Belgrade: Društvo za ETRAN, Beograd: Akademska misao (2022).

 ISBN 978-86-7466-930-3