Static Balancing Methods in Projection-Based Mesh Generation Algorithm

Sergej K. Grigorjev and Mikhail V. Yakobovskiy(✉)

Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Moscow, Russia
sergejgri@gmail.com, lira@imamod.ru

Abstract. The discussion is about parallel guarantied mesh-generation projection-based algorithm. Main subject of this article is load-balancing problem in distributed projection-based tetrahedral mesh generation algorithm. Algorithm is based on construction of triangle prisms, formed by orthogonal projection of base surface mesh. The advantage of using projection-based algorithm consists in guarantied tetrahedrisation of 3-dimensional domain. Main purpose of generated meshes consists in guaranteed detection of the topology of three-dimensional domains, which can be used for mesh adaptation algorithms.

Keywords: MPI · Mesh generation · Unstructured grid
Static load balancing · Triangulation · Rational numbers

1 Introduction

Geometrical calculating meshes are used in numerical modeling of solid environment. Generating such meshes is one of the hardest computational problems due to the large amount of time that is required to generate such type of mesh. One of ways of accelerating mesh generation is using parallel mesh generation algorithms.

Creating a parallel program creates an addition amount of tasks, which must be solved for parallel algorithm to become good. There are various problems, however, in this paper we consider problem of load balancing.

The main problem of load balancing for any parallel program is distributing the load of a parallel program over multiprocessor system uniformly [3]. There are two different classes of load balancing algorithms: static and dynamic. A static load balancing algorithm does not take into account the previous state or behavior of a node while distributing the load. On the other hand, a dynamic load balancing algorithm checks the previous state of a node while distributing the load, such as CPU load, amount of memory used, delay or network load, and so on [4].

M. V. Yakobovskiy—With the support of the RFBR grant 17-07-01604 A.

Each of these methods has their own advantages and disadvantages. Finding an optimal static load balancing is in general an NP-complete problem, unless some special cases [3]. It seems that optimal load balancing algorithm consists of good initial partitioning, which could be performed by some kind of static load balancing algorithm, and dynamic part, which redistributes load during the calculation. One of the principal costs of good initial distribution is expected to be the machine dependent cost of transferring the computational modules between processors [3].

In this particular paper we discuss static load balancing algorithm for guaranteed projection-based tetrahedral mesh generation algorithm. As this algorithm consists three major parts [1], each of them must have some kind of the algorithm. The fact that the considered algorithm tries to cover as much different surfaces as possible leads to a major problem: it is possible for one of computational modules on any of three main steps of algorithm take more time for processing, than all other modules together. This paper discusses load balancing for this type of algorithm is static load balancing algorithms. In this paper is discussed load balancing algorithms for parts of the algorithm, problems of creating balance weights for each computational module and some experiments with different size and topology objects.

2 Mesh-Generation Algorithm

For further explanation, let's point out main parts of algorithm. As input for this kind of algorithm we're using an oriented surface triangulation. For the purpose of simplicity, we suggest projection axis to be parallel to OZ. Therefore, all surface triangles are divided into 3 grand category, by the direction of z-component of their normal: TOP, WALL and BOT. Guaranteed mesh generation algorithm consists 3 main subsections: first part generates projection of BOT triangles onto TOP, triangulation of the resulting graph and creating a number of triangle prism, that covers the volume of the initial body; second - attachment of lateral faces of all faces with each other; and third - triangulating surface faces of each prism, so each prism is covered by surface triangulation. Any triangle prism is defined by six points: 3 points for each base prism triangle, even for degenerate case. Second important thing to admit is that algorithm uses rational numbers with arbitrary bit capacity for nominator and denominator in purpose of excluding any inaccuracies during the main calculation process, leaving them only to output part [1].

The first part of the algorithm can be described as follows:

1. [Cycle on i.] for $i = (1, N_t)$, for all TOP-triangles.
2. [Find nearest BOT-triangles.] For every i-th TOP-triangle, form an array of geometrically close BOT-triangles. Denote by N' the number of these triangles. Clear LAY substructure.
3. [Cycle on j.] Set $j = 1$. After the end of the cycle, go to step 8.
4. [Check hitting the projection-space.] Check the intersection of j-th triangle with the projection-space of i-th triangle. If the intersection was found, go to step 5, otherwise, go to step 7.

5. [Screen by TOP-triangles edges.] Execute screening of j-th triangle by edges of i-th triangle. Proceed to step 6.
6. [Screen by surface of BOT-triangles belonging to the projection-space.] Execute screening of the region constructed in step 5 by the surface of all triangles contained in LAY. Store the result in LAY.
7. [Termination condition for j cycle] If $j \le N'$, then set $j = j + 1$ and go to step 4, otherwise, go to step 8.
8. [Triangulate the projection.] Execute the algorithm of 2-dimensional triangulation on data stored in LAY. As a result, LAY structure contains N'' triangles.
9. [Cycle on k.] Execute step 10 for each $k = (1, N'')$
10. [Form prisms.] For k-th triangle from LAY, reestablish its projection in the plane of i-th triangle and the corresponding BOT-triangle. Save these six points, representing two triangles as a triangle prism.

Second part of the algorithm:

1. [Cycle on i.] For $i = (1, M)$.
2. [Find nearest prisms.] For i-th prism, form an array of geometrically close prisms. Denote by M' the number of such prisms.
3. [Cycle on j.] For $j = (1, M')$. After the end of the cycle, go to step 8.
4. [Cycle on k] Execute step 5 for $k = (1, 3)$, on the sides of i-th prism.
5. [Cycle on l] Execute step 6 and, if necessary, step 7, for $l = (1, 3)$, on the vertices of the upper base of j-th prism.
6. [Check hitting the plane.] Check the hit of l-th vertex in the plane of k-th side edge of i-ith prism. The result (0 or 1) is stored in his own cell in *count* array. If hit occurs, execute step 7.
7. [Inserting vertical edges into topology.] Check the hit of vertical edge of j-th prism on the side edge of i-th prism. If the hit occus, add intersection point and parts of this edge into topology of i-th prism.
8. [Calculate number of touches.] Sum the values of *count* array in *sum* variable. If $sum = 0$, go to step 9. If $sum > 0$, go to step 10.
9. [Checking intersection of side edges of edges of bases.] Find the intersection points of vertical edges of i-th prism with edges of bases of j-th prism. Add these points to topology of i-th prism, dividing corresponding edge into parts.
10. [Building intersection of side edges.] Check hit of parts of edges of bases of j-th prism on side edge of i-th prism. Add the corresponding elements to i-th prism topology.

Next important part of the algorithm is using arbitrary bit capacity for rational numbers. It provides exclusion of any calculation inaccuracies in this particular algorithm, but leads to a few issues. First issue, that happens while using such numbers, is an exponential growth of length for nominator and denominator. During this particular algorithm, however, the maximum length of nominator and denominator during the calculation is bounded above. To prove this, let's consider every part of the algorithm.

First part includes building intersections between 2 different initial surface triangles ribs. The equation for calculation intersection point between two lines is known:

$$\frac{x - x_1}{x_2 - x_1} = \frac{x - x_3}{x_4 - x_3} \tag{1}$$

which is taken from the canonical equation of the line. So the answer is:

$$x = \frac{x_1(x_4 - x_3) - x_3(x_2 - x_1)}{(x_4 - x_3) - (x_2 - x_1)} \tag{2}$$

In the worst case, each addition and subtraction increases the length of the nominator (3) and denominator (4):

$$n_2 = 2n_1 + 1 \tag{3}$$

$$n_2 = 2n_1 \tag{4}$$

Here n_1 is a bit capacity before operation, and n_2 - after it. So, based on Eq. (2) and using (3) and (4), maximum bit capacity for any coordinate would be for the nominator (5) and the denominator (6), n_1 is the bit capacity of the initial surface point coordinate, n_2 is a bit capacity of builded point. This is the maximum theoretical length of stored coordinates for the first part of the algorithm.

$$n_2 = 6n_1 + 3 \tag{5}$$

$$n_2 = 6n_1 + 2 \tag{6}$$

For second part of the algorithm, we are building same kind of intersection points, but initial points coordinates, which are used in (2) could be from points, which are created on a first stage of the algorithm. So, in this case the result maximum bit capacity for a point coordinate nominator in the worst case would be:

$$n_2 = 36n_1 + 21 \tag{7}$$

Second issue - is very hard to handle in terms of using MPI, is described further.

The most time-consuming part of the algorithm is docking prisms with each other, as it would be shown in practical section. Therefore, the main focus of this paper would be on the static load balancing algorithm of this particular part.

Problems of its load balancing for parallel distributed realization of this algorithm are main topic of this paper.

3 Load-Balancing Problem

3.1 General Parallel Realization Problem in Case of Load Balancing

Arbitrary bit capacity, while providing exclusion of any calculation inaccuracies, is very hard to handle in terms of using MPI. It is known, that using arbitrary bit capacity for rational numbers leads to exponential growth of length for nominator and denominator. And even fixed, a priory calculable maximum length is still much larger, than any standard type. And, as long as all points coordinates is also stored in such type, it is difficult to use MPI communications with it, and it leads to an increase in transmission time. Therefore, during the realization of any balancing algorithms it is necessary to minimalize number of MPI communications.

3.2 Load Balancing for the Projection Part

As soon as algorithm is positioned as guaranteed volume coverage algorithm, it is clear that under any TOP triangle could be literally any number of triangles. Furthermore, as is shown on Figs. 1 and 2, there could be other TOP triangles underneath it. On Fig. 1 on the left is shown initial volume, which needs to be filled, and part of the surface triangulation on the right. As it was mentioned in [1] the problem of too many BOT triangles fall underneath one TOP triangles could be solved in some cases by inserting an additional 0-thickness plates between TOP and BOT triangles. For example, in Fig. 2 underneath the pointed triangles we could place an additional inner plate so under each triangle of the mesh would be close in size number of triangles. But it does not work in example, shown on Fig. 1. For this kind of surface it is impossible to find place for plate, because there are no free space left for plate where we need to insert it (see Fig. 4).

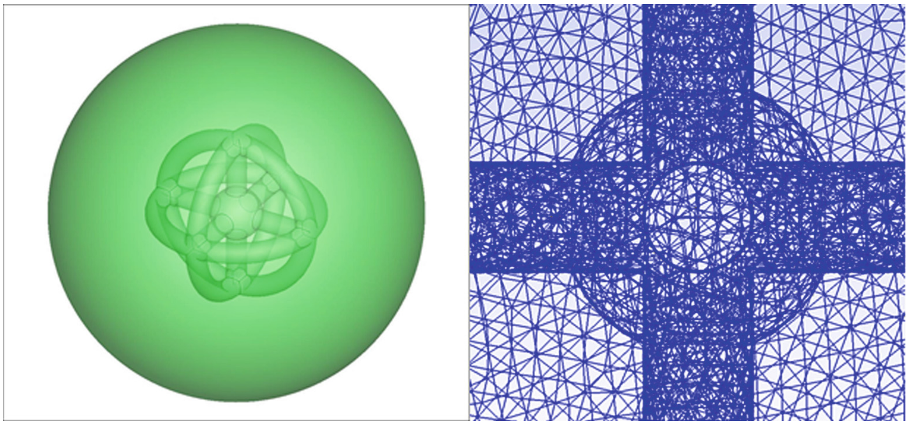

Fig. 1. Example of a surface with $N_p = 23068$ points and $N_t = 46174$ surface triangles

And, moreover, while we are creating a projection, we must check all triangles underneath considered triangle. But, during creating one projection it is unnecessary to communicate with other processes, so it is possible to create such initial triangles distribution, that at least some kind of load balancing could be achieved.

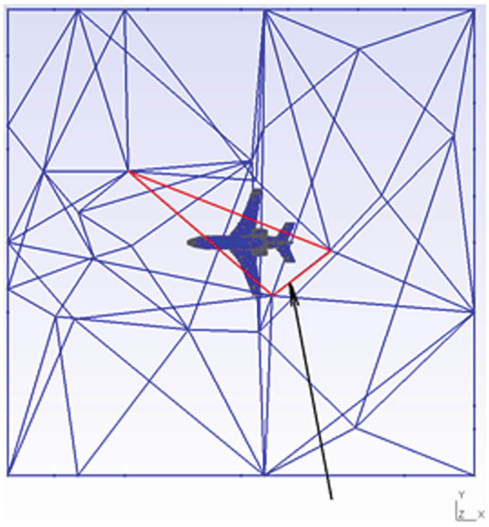

Fig. 2. Example of possible occasions for triangles mutual location

Creating a load balancing algorithm requires some kind of load weight, which could be used as weight for each element. As it is described in [1] using a 2-dimensional geometrical cache is necessary to decrease computational complexity for projection algorithm due to optimization of element search. As it is mentioned in [2], 2-dimensional cache can provide find all elements, which are close to the initial. So, as long as this type of cache could return all triangles that should be reviewed by projection algorithm, we can use number of BOT triangles, that is close to reviewed TOP triangle, as our balance weight.

But, on the other hand, such way concludes one major problem: if there are many colliding TOP and BOT triangles in the same cell of cache, One of the worst case scenarios is shown on Fig. 1 on the right. All surface of the cylinders, whose axis is parallel to the main projection axis, contains a large number of TOP and BOT triangles. That leads to one major problem: for all these TOP triangles their balance weight would be significantly higher, then for same triangles outside of the problem area. And, more than that, it is almost impossible to algorithmically find all BOT triangles, which would fall into projection onto current TOP triangle by any of their part (without, of course, creating full projection).

All this problem concludes into simple, but reasonably effective decision: for quasi-uniform surface mesh equal initial distribution leads into reasonable load balancing (the result would be mentioned below). Of course, this approach couldn't be used for other kind of meshes (for example, mesh on Fig. 2). It seems that solution for this problems, that concludes in good (by any criterion) balancing, requires and additional specific research.

3.3 Load Balancing for Attachment Part

This is the hardest part of the algorithm in terms of computational complexity and calculation time. The naïve way of balancing this part concludes into using just the same idea as in the first part. In that case, every triangle prism is represented as a triangle, which lays on OXY plane. This approach leads to two major problems. First, any prism is a 3-dimensional object. And 2 prisms could intersect with each other by a number of cases, which is shown on Fig. 3.

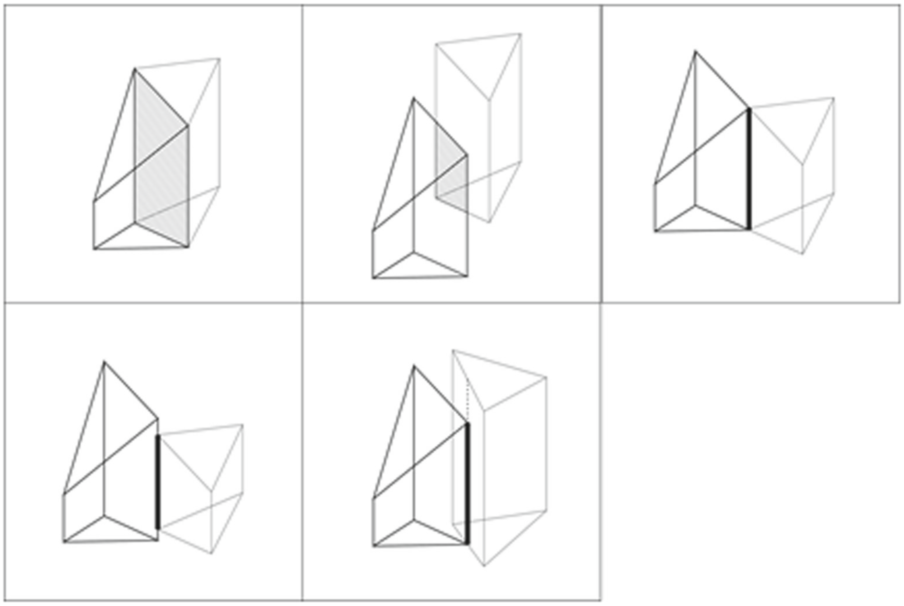

Fig. 3. Possible cases of collision of prisms, excluding the case with 0-thickness plates.

And, even more, they could be situation, when two or more prism are one above the other, without any colliding. Second, that partially follows previous paragraph, as long as docking of prisms is necessary, with every considered prism on a local process all its neighbor-prism must be stored, at minimum all 6 points, that define a prism. As we mentioned early, every point is stored as three rational numbers with arbitrary bit capacity. This concludes to extremely large memory usage, so it is necessary for the prism on each process to be as close to each other as possible.

This concludes us to main challenge: create a load balancing algorithm for prism docking, which guarantees as much domain integrity as possible and provides reasonable load balancing.

First of all, to separate whole array of prisms into any number of subdomains we need to create a graph, each vertex of this graph will represent a prism, and each edge should represent neighborhood relationship. While it is not known direct relationships between all prisms at this point of algorithm, we could use similar idea of creating of some kind of cache.

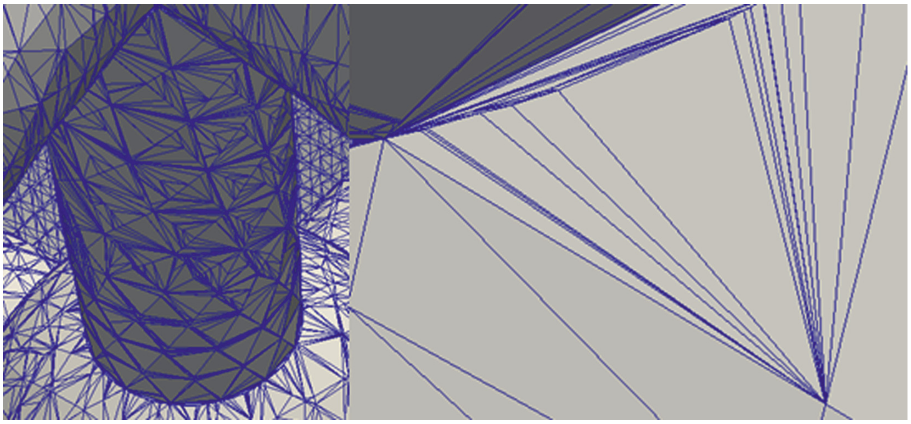

Fig. 4. Example with a large number of prisms laying in almost the same area.

On Fig. 4 is shown one of the worst case scenarios, when large amount of prisms are almost at the same place of space. On the left each triangle represents a surface for prism. On the right is shown much more closely the scale of problem in that particular case. This example leads to immediate conclusion: we cannot use 2-dimensional cache for this part of algorithm, because it would lead us directly to $O(N^2)$ operations, where N number of prisms To decrease this complexity as much as possible it is suggested to use 3-dimensional cache. There is second approach to resolving this problem, which would be described in Experiments-part.

With the same idea of caching with 2D-case, in 3D-case each cell of program cache represents a parallelepiped. For the simplicity of terminology, from here and below each cache cell would be named “cube”. This approach significantly lowers the amount of prisms that have fallen into each cube of cache. The most important usage of this idea consists in using the approximate number of neighbors for each prism as balance weight of prism, and this approximate neighborship relation for constructing edges for graph. Therefore it is formed a weighted graph, which is already distributed through the all processes. It is important, that this approach is not an ideal. For example, Fig. 5, where is shown resulting surface for body from Fig. 1. These topological artifacts, so called “stars”, cannot be optimized by any kind of geometrical cache.

Second trouble consists in separating this formed distributed graph on a number of domains, with minimum weight difference and least numbers of connections, which represents our demand to minimize the number of locally stored prisms. In purpose resolving this problem we use ParMETIS [5] for graph separation.

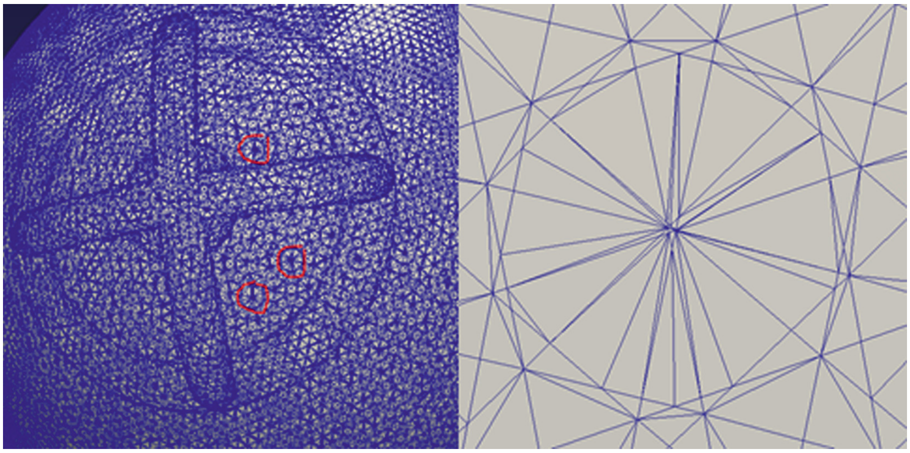

Fig. 5. Example of low efficiency of the 3D-cache.

Main problems of this algorithm consist this approach does not take in account time of inserting new vertices and edges into prisms, and does not consider any WALL triangles, that directly affect the calculation time.

3.4 Load Balancing for Triangulation of Prism Faces

It is the least expensive part of the algorithm. First thing that we have to mention that most of the prisms consists only 6 points, therefore for them it is necessary to add only 3 edge to each of them. Second thing that we should mention, that computational complexity for 2-dimensional triangulation depends mainly on number of vertices [2]. So, the main problem therefore is to decide, is it necessary to redistribute prisms after previous step of the algorithm or no.

The way of constructing previous weights for each prism is arguably inaccurate, so for each prism there would be a number of incorrect neighbors. Therefore, for the most prisms their weight is pretty much same.

This leads us to initial suggestion, that for purpose of load balancing of this stage of algorithm we could use distribution from previous stage. The advanced solution includes using number of vertices as a balance weight for this stage balancing.

4 Practical Experiments

From all of the experimental data we would consider at first all data about balancing load while docking the prisms. All experiments were done on 4 MPI-processes, on the one calculating node. All the data in Tables 1, 2 and 3 consists only time for docking of prisms.

First experiment is using surface, shown on Fig. 1. Total number of prisms in this experiment - 203 332.

Table 1. Experiment with different cache sizes, time for the second step of the algorithm. $Size_i$ – number of cells in that direction.

Dimension of cache and its size: $size_x * size_y * size_z$	Process 1, work time, minutes	Process 2, work time, minutes	Process 3, work time, minutes	Process 4, work time, minutes
2D-cache $100 * 100 * 1$	248.23	239.29	224.75	248.94
3D-cache $100 * 100 * 4$	188.62	235.84	230.21	212.18

The imbalance, which is shown in Table 1, is caused by the fact, that the balance weight value of each prism does not include the time of inserting new points and ribs, which are essentially created during this step of the algorithm. Increasing size of cache by 4 times caused further increase of imbalance from 10% in the first experiment to 20% in the second, but decreases calculation time.

Next idea for improving quality of balancing and further increasing load consists in rotating surface mesh around its center, so the least amount of prisms would have any additional points on their faces or edges; the result for this is shown in Table 3.

Table 2. Experiment with different cache sizes. Rotated surface.

Dimension of cache and its size: $size_x * size_y * size_z$	Process 1, work time, minutes	Process 2, work time, minutes	Process 3, work time, minutes	Process 4, work time, minutes
2D-cache $100 * 100 * 1$	190.87	190.25	200.87	191.14
3D-cache $100 * 100 * 4$	124.26	119.20	115.96	113.16

Simplification of prisms faces leads to significant increase of balance quality: now, first experiment provides almost only 5% of imbalance between processes, and second - near 9% of imbalance. Increasing imbalance while increasing cache size is provided by the fact that some prisms would eventually lose their balance weight, but the total number of new vertices and edges for each prism is constant, so for smaller prism there would be much less work to do, while for bigger prism there are still the same amount of work, which is represented by decrease of computational time.

The experiment shown in Table 3 shows, how initial surface triangulation of the same region will affect current algorithm (see Fig. 6). In this case the task is to fill inner part of these planes. As it is mentioned earlier, main difference is made by the amount of new vertices and edges, which are inserted into specific prism. Cache size for this experiment is $100 * 100 * 10$.

Now let's consider the last stage of the algorithm.

In Table 4 is shown test of hypothesis, which was described earlier about using initial distribution for previous stage. Surprisingly, the results are much

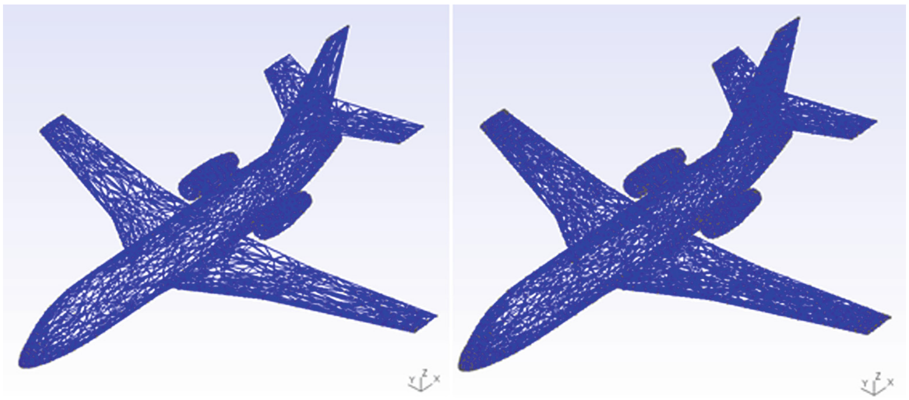

Fig. 6. Two different surface triangulations

Table 3. Experiment with the same initial surface, N_t – number of surface triangles.

Mesh parameters, number of triangles	Process 1, work time, minutes	Process 2, work time, minutes	Process 3, work time, minutes	Process 4, work time, minutes
Left mesh, $N_t = 6115$	92.052	98.204	96.409	117.529
Right mesh, $N_t = 10997$	253.704	269.194	296.033	295.033

better than they were to be expected. The weight imbalance in all cases is near 27%. This result is arguably bad, because it is still definitely high imbalance. But, for the same time, we can admit that even without creating any specific algorithm for this stage, using just what left from previous part, we already have a distribution of data that could be used for much easier data redistribution algorithm.

Table 4. Third stage load balance using the same distribution as in the second stage.

Mesh description	Process 1, work time, minutes	Process 2, work time, minutes	Process 3, work time, minutes	Process 4, work time, minutes
First example (Fig. 1)	25.122	20.413	29.575	23.325
First example (Fig. 1) rotated	24.73	18.414	22.509	18.122
First plane (Fig. 5, left)	11.01	11.028	12.182	16.382
Second plane (Fig. 5, right)	29.852	21.071	28.441	17.489

5 Conclusion

Load balancing is hard and important task for all parallel programs. During this work created static load balancing algorithm, which is based on three-dimensional cache for generating graph and uses ParMETIS library for separating graph. Reviewed main problems of static load balancing in application to guarantied mesh generation algorithm. It is proved, that using three-dimensional cache significantly decreases total calculation time, but increases load imbalance between each processes. Founded, that using initial balancing for second stage of the algorithm provides a reasonable load balancing for the third stage.

References

1. Grigorjev, S.K., Yakobovskiy, M.V.: Practical aspects of realization of projection tetrahedral mesh generation method. In: Proceedings of an International Scientific Conference on Parallel Computational Technologies (PaVT 2016), Arkhangelsk, 28 March–1 April 2016, pp. 499–504. Publishing Center of SUSU, Chelyabinsk (2016). ISBN 978-5-696-04801-7
2. Skvorcov, A.V.: Trianguljacija Delone i ejo primenenie [Delaunay triangulation and its application], 128 p. Tomsk State University, Tomsk (2002). ISBN 5-7511-1501-5
3. Iqbal, M.A., Saltz, J.H., Bokhari, S.H.: Performance tradeoffs in static and dynamic load balancing strategies. Technical report 86–13, NASA Langley Research Center, Hampton, VA (1986)
4. Shah, N., Farik, M.: Static load balancing algorithms in cloud computing: challenges and solutions. Int. J. Sci. Technol. Res. **4**(10), 365–367 (2015). ISSN 2277–8616
5. Karypis, G.: METIS and ParMETIS. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1117–1124. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4. ISBN 978-0-387-09766-4