Parallelization of Finite-Volume Numerical Methods of Computational Fluid Dynamics by Means of Shared Memory Computing Systems

Anastasia Gulicheva(✉)

MIREA – Russian Technological University, Vernadsky Avenue 78, 119454 Moscow, Russia
gulicheva@mirea.ru

Abstract. The article considers the problems of parallelization of finite-volume numerical methods concerning the calculation of the streaming flow of conservative quantities through the boundaries of computational cells for systems operating with shared memory. Three approaches to solving this problem were analyzed, for which a theoretical justification was carried out, implementation, testing, analysis of the performance and scalability of the proposed methods were performed.

Keywords: Numerical Methods · Finite Volume · Parallelization · Shared-Memory Systems · Flows of Conserved Quantities · Computational Grids

1 Introduction

At present, there is a tendency in the world to develop supercomputer cluster systems, the computing nodes of which consist of a large number of cores and are capable of performing calculations with shared memory using a huge number of threads. Efficient use of such computing nodes becomes a priority. When performing calculations using finite-volume numerical methods, calculations require performing two phases of calculations at each iteration of calculations. At the first phase, within each computational cell, independent of the remaining cells, local physical quantities are recalculated (this phase can be ideally parallelized due to the independence of the calculations). During the second phase, the computational cells exchange information with each other by following streams of conservative quantities between them. At this phase, conflicts may arise over access to the data of computational cells, and the analysis of effective methods for parallelizing calculations at this phase is the subject of this work.

The main objective of this article is to implement computation parallelization using the graph coloring technique for conflicts arising from the streaming flow of conservative quantities of unstructured surface computational grids and for ordinary regular volumetric grids, the cells of which are rectangular parallelepipeds.

In the parallel calculation of physical quantities and the flow of their streams between cells on a shared memory computer, we encountered the following problem: in the parallel calculation of the flow of a quantity flow through two boundaries incident to the same cell, there is a conflict in access to the data of this cell. Therefore, for a correct calculation, such conflicting flows should be calculated strictly sequentially. To solve this problem, several methods of parallel computation of streams of conserved quantities were implemented: using OpenMP, using intermediate storage of data at cell boundaries, and using conflict graph coloring.

2 Problem Statement

In the calculations performed within the cells, physical quantities of two types are used (for example, the problem of gas dynamics): *primitive* (which characterize the state of matter) and *conservative* (for which conservation laws are satisfied).

$$\vec{W}\ \textit{vector of conservative variables} \left(= [\rho,\ \rho u,\ \rho v,\ \rho w,\ \rho E]^T\right),$$

$$\vec{W}_p\ \textit{vector of conservative variables} \left(= [\rho,\ \rho u,\ \rho v,\ \rho w,\ \rho E]^T\right).$$

Therefore, we obtain a system characterized by vectors of conservative variables $\overrightarrow{W}$, flow vectors $\overrightarrow{F}_c$ and $\overrightarrow{F}_v$, as well as original member $\overrightarrow{Q}$ extended equations of the form $(N-1)$ [1]. The vector of the conservative variables now looks like:

$$\overrightarrow{W} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \\ \rho Y_1 \\ \vdots \\ \rho Y_{N-1} \end{bmatrix}.$$

The state of the substance in the cell is fully characterized by both the vector of primitive variables and the vector of conservative variables, and each can be obtained from the other using the corresponding transformation.

Each computational cell is adjacent to other computational cells, adjacent cells touch through a common boundary (for volumetric meshes, the boundary is a flat face, as in Fig. 1(b), for surface meshes, the cell edges are the boundary, as in Fig. 1(a)) through the common boundaries there is an overflow of flows expressed in conservative values.

In Fig. 1, D – primitive quantities, U – conservative quantities, and F – flows.

The general scheme of calculation and substance exchange for one cell in a single iteration can be outlined as follows:

1. calculation of primitive values D in the calculation cell;
2. obtaining a vector of conservative values from the vector of primitive values, that is, $D \rightarrow U$;

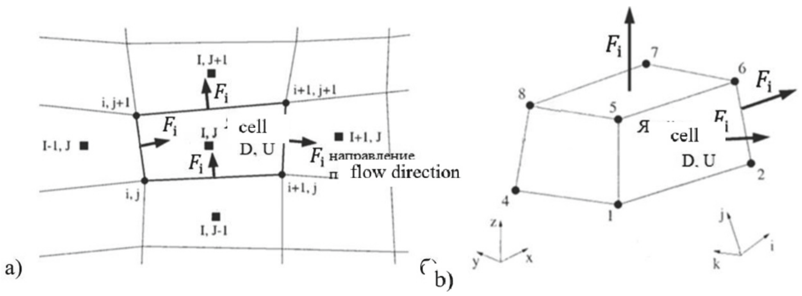

Fig. 1. Examples of cells in one-dimensional (a) and two-dimensional (b) grids.

3. for each boundary of the considered cell, the matter fluxes F_i are calculated, expressed in conservative values;
4. recalculation of the vector of conservative values taking into account flows through the boundaries $U = U + F_i S_i \Delta t$ for all boundaries;
5. obtaining a vector of primitive values from the vector of conservative values, i.e. $U \to D$.

When all flows from step 4 are calculated simultaneously, a conflict arises over access to the vector of conservative values of the cell. Specifically, all the streams that pass through boundaries incident to the same cell are involved in pairwise conflicts [8].

3 Parallelization of Computing Flows of Conserved Values

When using multi-threaded machines, there is a desire to perform thread recalculation in parallel mode. To ensure this, it is necessary to resolve conflicts on access to conservative values when they are corrected due to the flow of matter flows, and this can be resolved by implementing the following methods:

1. Using the OpenMP critical sections mechanism. Cons of the approach: the need to use critical sections, which can be problematic for some architectures (for example, for Elbrus).
2. Saving flows first at all boundaries, and then a one-time recalculation for each cell. To implement this mechanism, it is also necessary to store information about all boundaries that are input (through which flows enter) and output (through which flows exit) for the cell. The sequence of actions in this case:
   a. bypass all borders between cells, calculate the flows on them and save them on the borders;
   b. bypass all cells of the computational grid to the vector of conservative values, add the corresponding fluxes from the input boundaries (corrected for area and time step) and subtract the fluxes from the output boundaries.

   Cons of the approach: the need for each cell boundary to store information about whether this boundary is an input or an output; the need to maintain conservative values of flows at the boundaries.

3. Dividing the set of boundaries between the calculated cells into subsets in such a way that each subset does not contain conflicting boundaries. Cons of the approach: additional steps are needed to split the set of boundaries into subsets without conflicts.

3.1 Reduction of the Parallelization Problem

The graph is entered. The vertices of the graph are the boundaries of the computational grid. Two graph vertices are connected by an edge if the corresponding mesh boundaries conflict. Next, we solve the problem of vertex coloring of the resulting graph. To do this, we introduce a definition.

Definition 1. Let G be an undirected graph. A vertex coloring of a graph G is a mapping f: $(G) \to \mathbb{N}$ such that $(v) \neq (v)$ for all $\{v, w\} \epsilon (G)$.

The numbers $f(v)$ are called the colors of the vertices v. In other words, sets of vertices with the same color (value f) must be independent sets or matchings, respectively. We are interested in using as few colors as possible [7].

The value of the optimum in the vertex coloring problem (that is, the minimum required number of colors) is called the chromatic number of the graph [11].

Theorem 1. Let G be an undirected graph with maximum degree k. Then there is a vertex coloring of the graph G, where no more than $k + 1$ colors, and this coloring can be found in linear time.

Theorem 2 (Brooks). Let $G = (V,)$ be a connected graph that is not a complete graph in which the greatest degree of a vertex is $d \geq 3$. Then the graph can be colored with d colors.

In this article, we use a greedy algorithm. Using it, the vertices $v_1, \ldots, v_n$ are ordered and sequentially assigned to the vertex v_i the smallest available color that was not used to color the neighbors vi among $v_1, \ldots, v_{i-1}$, or adds a new one. The quality of the resulting coloring depends on the chosen order.

In this work, we considered two variants of the greedy algorithm. The difference lies in the fact that in the first case, any arbitrary vertex is selected and assigned the first color, while in the second case, the first color is already assigned to all vertices of the graph. Subsequently, the greedy algorithm starts working in both cases.

3.2 Application of Graph Coloring in Parallelization

Let's declare a graph. The vertices of the graph are the boundaries of the computational grid. Two graph vertices are connected by an edge if the corresponding mesh boundaries conflict. Next, we solve the problem of vertex coloring of the resulting graph.

In this article, we used graph (Fig. 2) to show the results.

In Figs. 2 and 3 we see that the maximum degree of a vertex is four, which means that such a graph can be colored in 4 colors according to theorem 2, this ensures planarity [9].

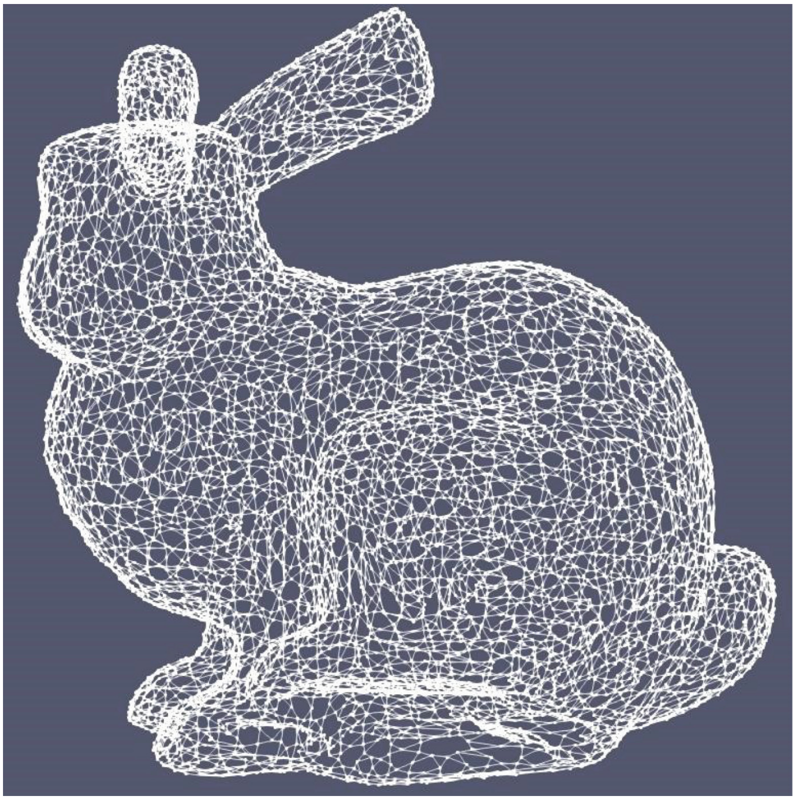

Fig. 2. An example of an unstructured surface mesh “Rabbit”.

3.3 Optimized Greedy Coloring Method

Accordingly, the greedy coloring algorithm produces a result of 5 colors. Here, the first random vertex is selected, which is assigned the value of the first color, and then all subsequent vertices are colored in such a way that the color does not match the already colored vertices with which this vertex has a common edge. The color is selected from the already used colors, but if it is impossible to use them, a new color is added.

But we have developed a more optimal method, and when we run it, the result is: 4 colors. Now we describe the coloring algorithm in 4 colors [3, 10].

1. Since the degree of each vertex of the graph is at most 4, it is possible to paint in 5 colors in a greedy way. After that, we will try to recolor all the vertices painted in color 5.
2. Consider a vertex with color 5. If it cannot be recolored, then all its 4 neighbors are colored 1, 2, 3, 4 (Fig. 4).
3. If it is possible to recolor at least one neighbor in a color different from 5, then the vertex under consideration can then be recolored. This means that in the worst case, no neighbor can also be repainted, which means that the neighborhood of the considered vertex has the form, as shown in Fig. 5.

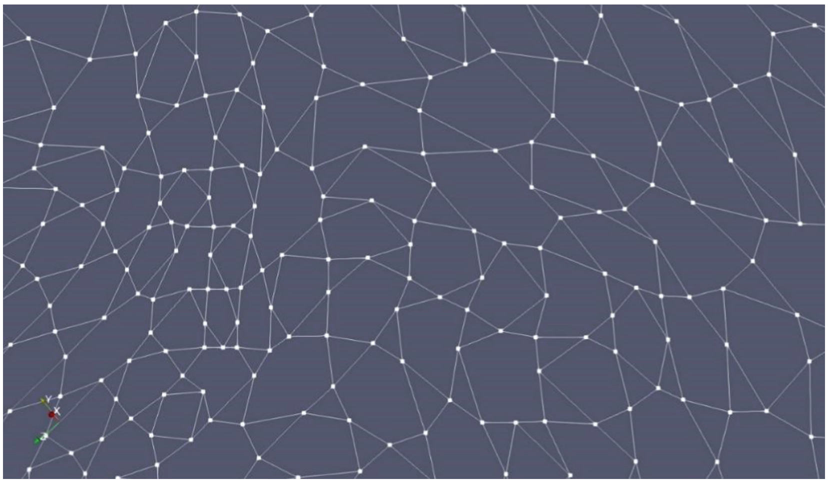

Fig. 3. An example of a close-up view of an unstructured surface mesh “Rabbit”.

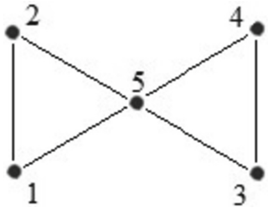

Fig. 4. An example of a case of coloring in 5 colors.

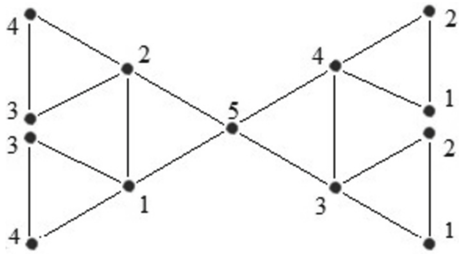

Fig. 5. Worst case coloring in 5 colors.

4. It follows from the appearance of the neighborhood that color 5 can migrate in any direction, for example, exchanging places with 4, as shown in Fig. 6.
5. With the help of the described color 5 migration mechanism, it is possible to drive all such colors to the graph boundary (where the degree of vertices is less than 4) and recolor in a greedy way. Since we are considering the worst-case scenario, we will assume that the graph has no boundaries.

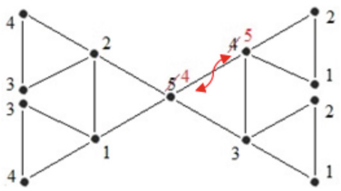

Fig. 6. Color replacement example.

6. If there is more than one vertex of color 5 in the graph, then using the migration mechanism, we bring them closer to each other in the following configuration, as shown in Fig. 7.

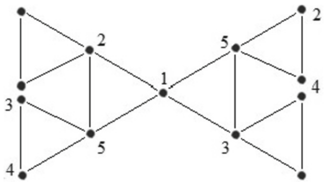

Fig. 7. The result of the migration.

After approaching two vertices of color 5 at a distance of 2 to each other, in this case we can recolor them to color 1, and recolor one to 5, thus reducing the number of vertices of color 5. Since we are considering the worst case, we will assume that we have one vertex of color 5 left in the graph.

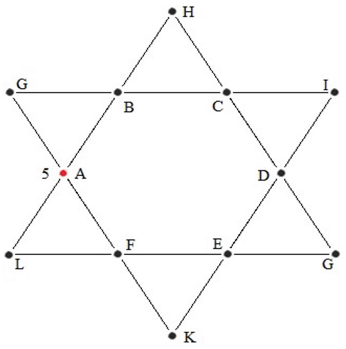

Fig. 8. An example of an even length loop.

7. Let's assume that there is only one vertex of color 5 left in the graph without a border, and no recoloring of this vertex can be done with any actions to migrate this vertex. Consider one of the larger cycles that this vertex (A) enters, as shown in Fig. 8.

Let this cycle be of even length. For definiteness, let's draw a cycle of length 6, as shown in the figure. Since the vertex A cannot be recolored, without loss of generality we will assume that the vertices G and B are colored in colors 1 and 2 (which of them is colored in color 1 and which in color 2 does not matter), and the vertices L and F are colored in colors 3 and 4. We agreed to assume that after the migration of color 5 to vertex B, this vertex B cannot be recolored. Since, after the migration of color 5 to vertex B, the vertices AG will be colored in colors 12, then the edge HC must be colored in colors 34 (we denote HC ~ 34).

By analogous reasoning, we get DI ~ 12, EG ~ 34, FK ~ 12. However, FK ~ 12 contradicts LF ~ 34. This means that at some point during the migration of color 5 along an even cycle, we could recolor the vertex. This can be seen in Fig. 9. Let us similarly consider the case when the large cycle is of odd length. To be specific, let's draw a cycle of length 5 as shown in Fig. 10 below.

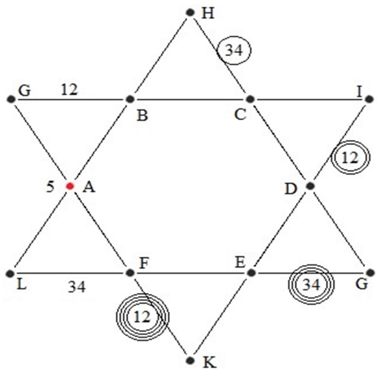

Fig. 9. An example of a contradiction in a cycle of even length.

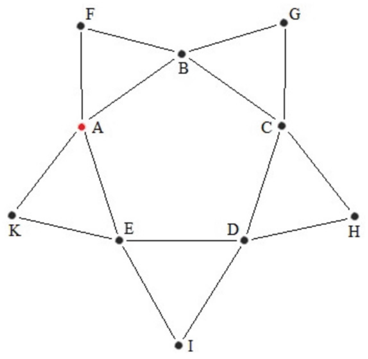

Fig. 10. An example of an odd length loop.

And once again, we start reasoning from the fact that since vertex A cannot be recolored. Without loss of generality, let's assume that vertices F and B are colored with colors 1 and 2. It doesn't matter which one is colored with color 1 and which one with color 2. Similarly, vertices K and E are colored with colors 3 and 4. We agreed to assume that after the migration of color 5 to vertex B, this vertex B cannot be recolored. Since after the migration of color 5 to vertex B the vertices AF will be colored in colors 12, then the edge GC must be colored in colors 34 (we denote GC ~ 34). By similar reasoning, we get DH ~ 12, EI ~ 34, AK ~ 12, which contradicts the fact that KE are colored in colors 3 and 4. This means that at some point during the migration of color 5 along an odd cycle, we could recolor top. The result is in Fig. 11.

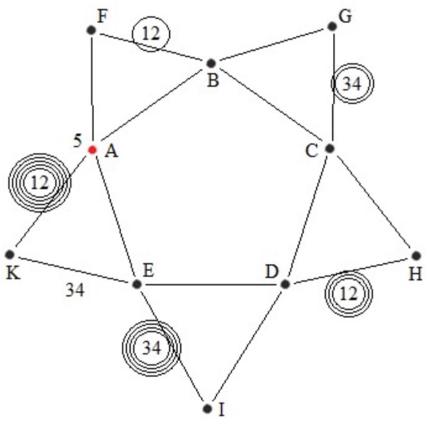

Fig. 11. An example of a contradiction in a cycle of odd length.

3.4 Verification of the Algorithm

The result of the algorithms described above is shown in Fig. 12, where you can see that the presented graph can be colored not in 5 colors, but in 4. The vertices that have two colors are those vertices that have been repainted, they have an outer rim - this is the color that the greedy algorithm produced, the central color is the color in which they can be recolored to implement a more optimal coloring.

4 Discussion

To measure the scalability of computations on an unstructured surface computational grid, we used a test surface of a streamlined three-dimensional body containing about 105 nodes and 105 cells. In the cells, calculations were performed related to modeling the flow of a liquid film, solving the heat balance equations on the surface, as well as restructuring and smoothing the surface (Fig. 13).

The calculations were performed on one computing node. The main goal of the launches was to measure the strong scalability of computing using a different number of threads within the computing node [2, 5, 6].

That is, the same surface was used for all launches (which was provided by a different number of threads).

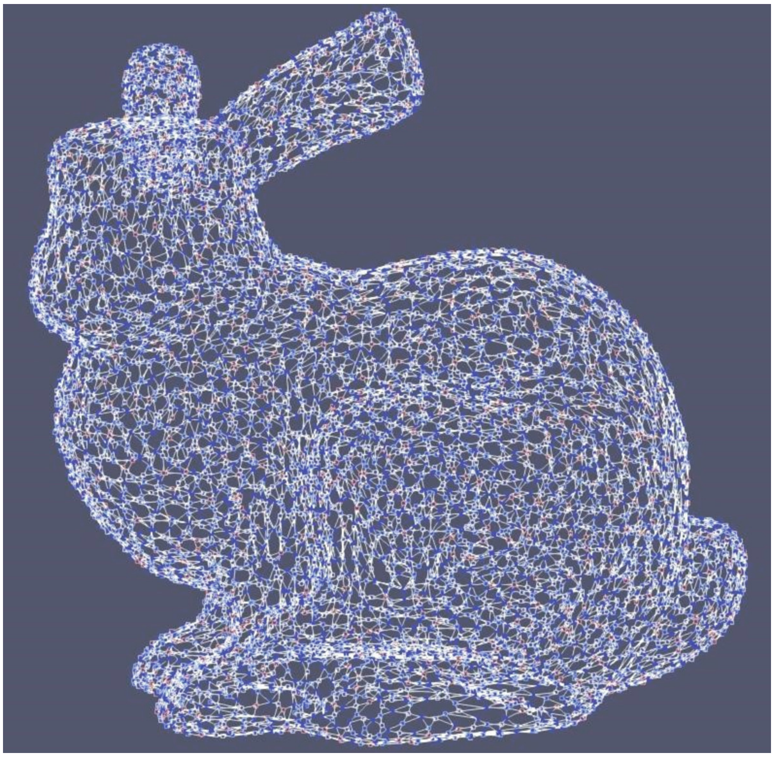

Fig. 12. The result of optimizing the greedy algorithm on the "Rabbit" graph.

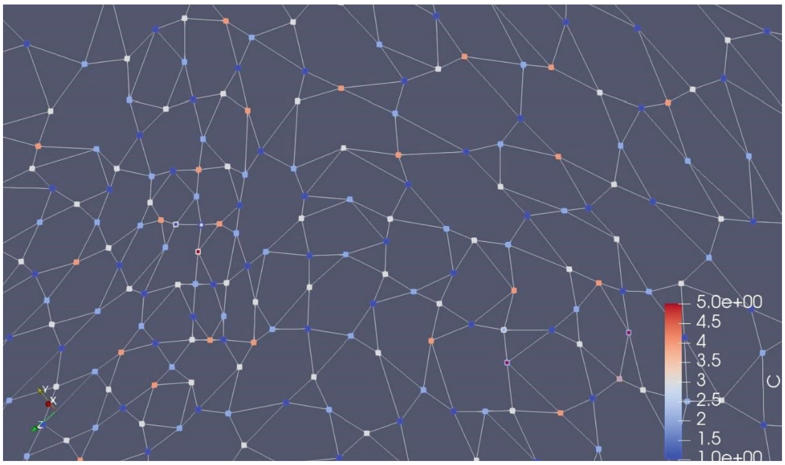

Fig. 13. The result of the optimization of the greedy algorithm on the "Rabbit" graph near.

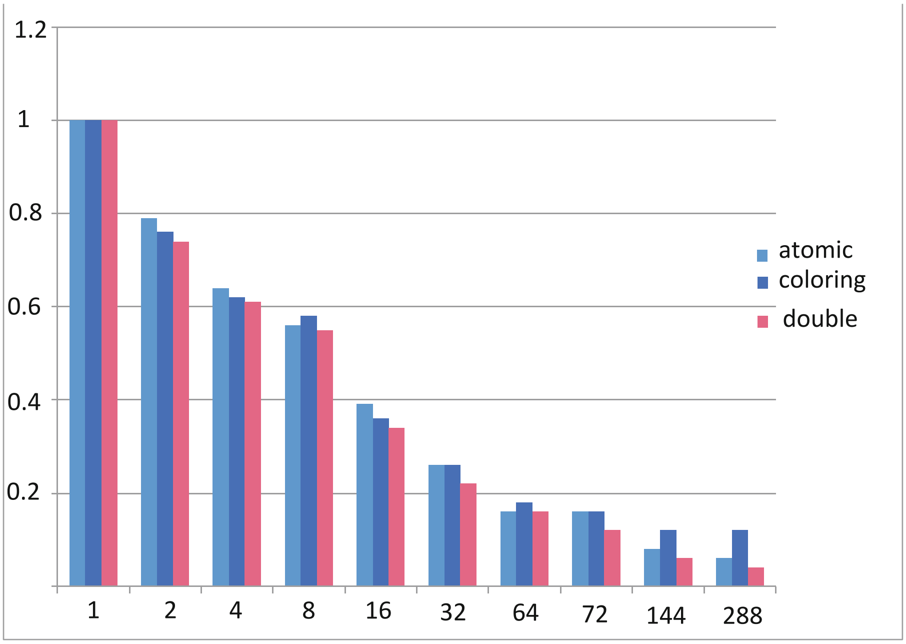

Fig. 14. Efficiency of computing scaling on a single computing node based on the Intel Xeon Phi KNL microprocessor microprocessor with an increase in the number of threads.

Figure 14 shows a diagram of the efficiency of computing scaling for various methods of parallelizing thread computations with an increase in the number of threads within a single node. At the same time, the value of

$$s(i) = t(1)/i(1),$$

where $t(1)$ is the reference time, was taken as acceleration at the number of nodes equal to i. In this case, the computational scaling efficiency is understood as the value of

$$e(i) = s(i)/i.$$

Its physical meaning is as follows. It can be assumed that with an ideal parallelization of calculations, with an increase in the number of threads by a factor of n, the execution time decreases exactly by a factor of n. Thus, in the case of ideal parallelization, $(i) = i$, and $(i) = 1$. The scaling efficiency of computations is a convenient indicator of the quality of creating an executable parallel code and comparing different computing systems with each other [4]. Note that superlinear scalability is quite possible (when the value of (i) becomes greater than 1 (Table 1).

Thus, we can conclude that with an increase in the number of computational threads, the most efficient way is to divide edges (borders between computational cells) into independent sets (graph coloring) and process them in turn.

Table 1. Calculation scaling efficiency indicators for various methods of thread computing parallelization with an increase in the number of computing nodes used.

Number of threads	atomic	coloring	double
1	1	1	1
2	0.79	0.76	0.74
4	0.64	0.62	0.61
8	0.56	0.58	0.55
16	0.39	0.36	0.34
32	0.26	0.26	0.22
64	0.16	0.18	0.16
72	0.16	0.16	0.12
144	0.08	0.12	0.06
288	0.06	0.12	0.04

References

1. Vorobyov, E.S., Vorobiev, V.E.: Numerical methods and mathematical modeling. Fundamentals of numerical methods and techniques for building mathematical models based on them and these solutions in various packages, Kazan (2017)
2. Rybakov, A.A.: Distribution of the computational load between the nodes of a heterogeneous computing cluster. Progr. Products Syst. Algorithms **1**, 1–7 (2018)
3. Golovchenko, E.: Overview of Graph Decomposition Algorithms. Preprint IPM № 002 (IPM named after M. V. Keldysh) (2020)
4. Rybakov, A.: Internal representation and mechanism of interprocess exchange for a block grid for supercomputing. Software products and systems. Algorithmiya **8**(1), 121–134 (2017)
5. Software Developer's Guide for Intel 64 and IA-32 Architectures (Intel Corp.), Consolidated Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 (2019)
6. Jeffers, J., Reinders, J., and Sodani, A.: High-Performance Programming of Intel Xeon Phi Processors, Knights Landing Edition. Morgan Kaufmann, Burlington (2016)
7. Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures, pp. 138–144 (2009)
8. Blazek, J.: Computational Fluid Dynamics: Principles and Applications. Elsevier, Amsterdam (2001)
9. Molloy, M., Reed, B.: Graph colouring and the probabilistic method. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-04016-0
10. Korte, B., Vygen, J.: Combinatorial Optimization Theory and Algorithms. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-56039-6
11. Anferov, M.A.: Algorithm for searching subcritical paths on network graphs. Russian Technol. J. **11**(1), 60–69 (2023)