HAL open science

Software transactional memory for parallel meshing applications

Isaïe Muron, Franck Ledoux, Cédric Chevalier

To cite this version:

Isaïe Muron, Franck Ledoux, Cédric Chevalier. Software transactional memory for parallel meshing applications. COMPAS 2025 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jun 2025, Bordeaux, France. ⟨hal-05121848⟩

HAL Id: hal-05121848

https://hal.science/hal-05121848v1

Submitted on 20 Jun 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Autorisation HAL

HAL Authorization

Compas'2025 : Parallélisme / Architecture/ Système
Centre Inria de l'université de Bordeaux - Bordeaux - France, 24-27 juin 2025

Software transactional memory for parallel meshing applications

Isaïe Muron*, Franck Ledoux, Cédric Chevalier

CEA, DAM, DIF – F-91297, Arpajon - France
Université Paris Saclay, CEA, LIHPC – 91680 Bruyères-Le-Châtel - France
isaie.muron@cea.fr

Abstract

Parallelizing remeshing algorithms presents significant challenges due to their dynamic nature, where operations alter data that dictates subsequent actions. We aim to address problems created by these dynamic workloads without relying on conflict avoidance, and propose an alternative approach to ensure data integrity in this context. To that end, we leverage specific data structures, synchronization mechanism and programming language features. We benchmark our implementation with two algorithms: a topology-heavy kernel with non-trivial parallelization, and a simpler vertex relaxation kernel. Preliminary results suggest STM effectively manages conflicts, maintaining data validity under concurrency. Our work offers a robust framework for parallel remeshing, balancing correctness and performance.

Keywords: data structure, transactional memory, remeshing, dynamic workload, correctness

1. Introduction

1.1. Context

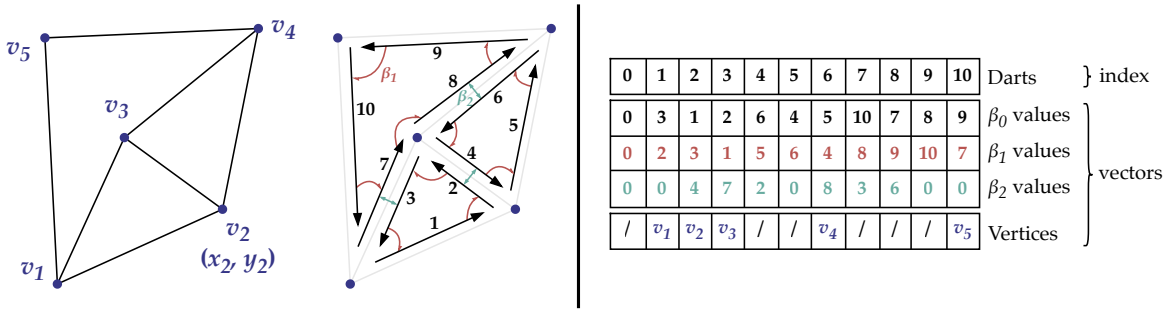
Many scientific applications today rely on solvers using a discrete decomposition of the problem space to compute a solution. We focus on the generation of that discrete decomposition, commonly referred to as the mesh. Larger, finer, and higher quality meshes improve subsequent simulation results, but require more resource to generate. We aim to provide an efficient solution for massively parallel mesh generation and remeshing on modern hardware. However, this is not trivial: meshing applications often involve dynamic problems, where each operation affect data used to decide how and if the next operation is applied, meaning the workload, and conflicts, cannot be computed beforehand. We present our approach to guarantee data integrity in the context of dynamic meshing algorithms, based on the usage of combinatorial maps, Rust and Software Transactional Memory (STM).

1.2. Combinatorial maps

Our implementation uses formal definitions of previous works [1, 2]. This paper will only briefly introduce the structure to better contextualize our contribution. More information can be found in examples and manuals of current implementations [3, 4].

Combinatorial maps of dimension N, or N-maps, provide a combinatorial data structure where topology is described using functions defined over base elements called darts. Geometry is also described by defining functions which associate vertices (or any other data) to subsets of darts. Figure 1 shows a basic example of 2D mesh representation using a 2-map.

* Corresponding author



0	1	2	3	4	5	6	7	8	9	10	Darts
0	3	1	2	6	4	5	10	7	8	9	β_0 values
0	2	3	1	5	6	4	8	9	10	7	β_1 values
0	0	4	7	2	0	8	3	6	0	0	β_2 values
/	v_1	v_2	v_3	/	/	v_4	/	/	/	v_5	Vertices

Figure 1 – 2-map representation of a heterogeneous 2D mesh

An N-map contains a set of beta functions defining topology, where the β_i function can be interpreted as an application which enables i-cell traversal. For example, the β_1 function allows us to move through darts modeling different edges while the β_2 function allows us to move across darts modeling different faces. The β_0 function is an optional extension of the structure defined as the inverse of β_1: $\beta_0 \circ \beta_1 = I_d$.

We implement all basic operations defined on N-maps, including orbit computation, a specific definition of dart subsets which corresponds to usual mesh elements: for example, the face associated with dart d_7 corresponds to the orbit $o_{\beta_1}(d_7) = \{d_7, d_8, d_9, d_{10}\}$ (i.e. all darts accessible from d_7 using compositions of β_1), while the edge associated with d_7 is the orbit $o_{\beta_2}(d_7) = \{d_7, d_3\}$. These are used to bind arbitrary data to the map. The binding logic is referred to as embedding; it is what allows us to define coordinates and model meshes.

The integrity of the structure is maintained through the usage of STM. We choose it for its composability as well as the guarantees it provides in concurrent contexts.

1.3. Software Transactional Memory

Dynamic problems form a category of problems where a given action may alter data used to determine how or if the next action should be taken. This leads to the inability to predict data allocation, movement, and access conflicts in practice.

In order to guarantee correctness when solving such problems, different solutions are available. Usual approaches can involve a form of conflict avoidance such as coloring or partitioning [5, 6], as well as the usage of aggressive locking mechanisms as a fallback. This is a pessimistic approach as the algorithm may not progress until data is locked, and correctness is secured.

Pessimistic approaches may lead to scalability issues due to useless waits, preventing execution of operations that would not ultimately conflict. We instead choose Software Transactional Memory [7, 8] to handle data structure synchronization. Unlike lock-based approaches, STM is optimistic in the sense that the algorithm does no wait for locks to run computations. While STM has been used for simulation [9] and meshing operations have been included in STM benchmarks [10, 11], the mechanism hasn't been leveraged for extensive meshing applications.

Transactions serve as wrappers for synchronization-sensitive routines. Code inside a transaction is run using a separate registry to keep track of reads and writes, making no edits to the actual data. Once completed, data is locked to check if it still is consistent with the state it was in during computation. If it is, changes are published and data is written from the alternate registry to memory, before locks are released. If the data has been edited since its use, the alternate registry is discarded, and no edits are committed. The transaction is then retried or cancelled.

We can use git remote repository synchronization as an analogous example. After some work on a local instance of the repository, the user registers changes under the form of a commit. If the remote repository has seen changes since the local copy was made, git will enforce a first pull before publishing local changes (push). This corresponds to the verification of transactions. If the changes do not conflict, nothing will be asked of the user. However, if remote and local changes conflict, pushing local changes will be blocked until conflicts are resolved.

2. Benchmarked algorithms

Our work is open-source, the implementation of core structures, kernels and benchmarks can be accessed on GitHub [12]. Details regarding the implementation are included in Appendix A.

We use two algorithms to benchmark our *N*-map-based mesh data structure. The first, *cut-edges*, is a topology-heavy kernel with dynamic workloads, making its parallelization non-trivial. The second, *shift*, is a simple vertex relaxation routine which would usually be parallelized using graph coloring or partitioning.

2.1. Algorithms presentation

The *cut-edges* benchmark takes a triangular mesh as input and iterates until a length-based predicate is verified by all mesh edges. Given an edge e and its length L_e , if L_e is greater than a target length L_{thresh}, e is cut in half, and two new edges are inserted from the new vertex, located at the midpoint of e, to vertices of triangles on each side of the edge. Figure 2 illustrates the operation on both the mesh and our data structure.

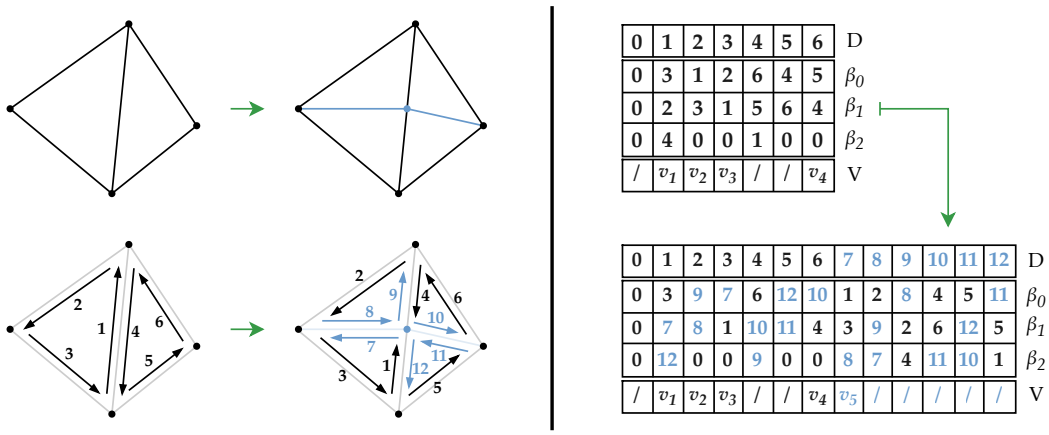


Figure 2 – Operation of the *cut-edges* benchmark

The operation involves adding new darts to the structure, editing the existing topology, and inserting a new vertex. Note that this operation can and is executed on the boundary of the mesh. In this case, only a single edge is inserted, using three new darts instead of six.

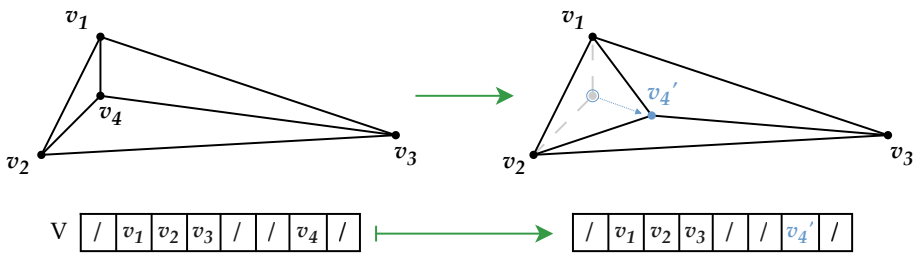


Figure 3 – Operation of the *shift* benchmark

The *shift* benchmark focuses on geometrical operations and does not affect topology. Given a mesh, all inner vertices are moved to the average of their neighbors' positions. The operation is shown for a single vertex with three neighbors in Figure 3. Unlike the previous algorithm, the workload is not dynamic; it can be parallelized using conflict avoidance methods (e.g. coloring). This makes the benchmark a good anchor for future comparisons against usual solutions.

2.2. Conflict resolution

Executing both algorithms described above in parallel introduces conflicts that can be fixed using synchronization mechanisms.
In the *shift* benchmark, conflicts may occur when two neighboring vertices are processed concurrently: if the first's new value is written after the second fetched the old value for its own computation, the second vertex's new value will be computed using the previous, now invalid position of its neighbor.
In the *cut-edges* benchmark, conflicts occur when two edges sharing a face are processed concurrently.

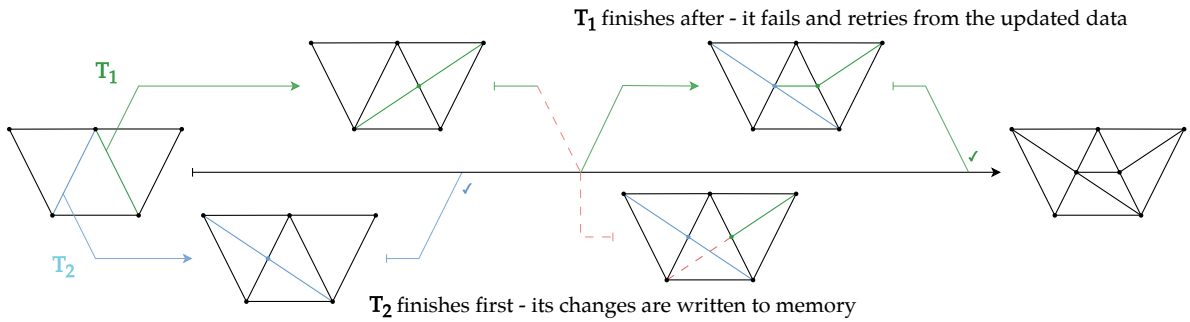


Figure 4 – Execution flow over conflicting operations of *cut-edges*

STM allows detection and resolution of the conflict presented above. Figure 4 illustrates a possible execution flow where the processing routine of each edge is encapsulated in a transaction. The transaction that completes first commits its changes to the structure. When the second completes, changes to the original data are detected and trigger a retry, leading to computation of the valid result.

3. Results and Analysis

We run each benchmark on a single 2-socket node using two AMD EPYC 7763 processors, totaling 128 physical cores and 256GB of RAM. We refer to cores and threads interchangeably as we spawn at most one thread per core. Our kernels are parallelized using the `rayon` crate [13], which implements parallel iterators and work-stealing in shared-memory contexts. Parallelization is done over the workload: `rayon` allows us to define parallel iterators over the collection of cells to process (vertices and edges, respectively in *shift* and *cut-edges*), akin to OpenMP's `parallel for` clause. We do not use any form of partitioning or coloring, and instead let STM handle conflict resolution as described previously.

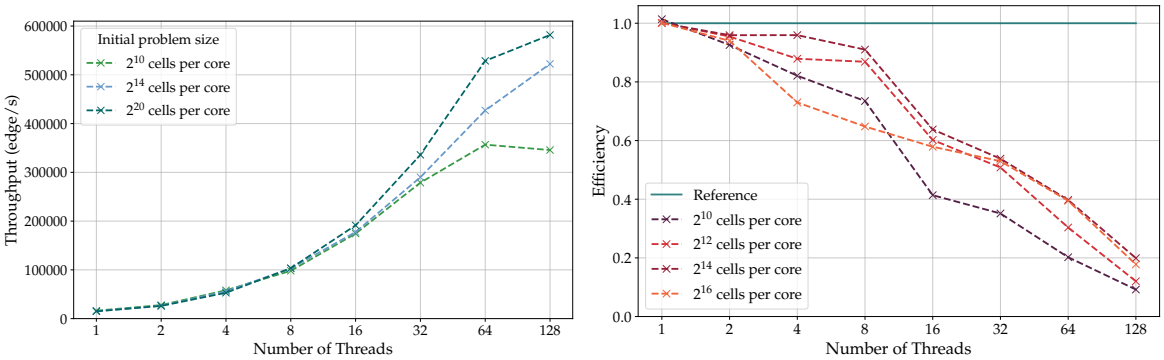
We measure performance using strong and weak scaling studies: strong scaling involves processing a fixed-size problem with an increasing number of threads, while weak scaling involves processing increasingly large problems to keep the workload per thread constant.

We compute the usual associated metrics for the *shift* benchmark, respectively speedup S and efficiency E, using the same formula: $T_{exec}(n_{threads} = 1)/T_{exec}(n_{threads} = N)$. We fix ideal values using N threads to $S_{ref} = N$ and $E_{ref} = 1$; this choice is detailed in Appendix B. We do not compute speedup and efficiency for the *cut-edges* benchmark due to the exponential variation of the workload during execution. Instead, we compute its throughput defined as the number of time the operation illustrated in Figure 2 is accomplished per second, i.e. the number of edges processed per second.

The *shift* routine is run for 100 rounds to reach significant execution time and accurate measures. Averages and standard deviations are computed using rounds as the sample sets, while we measure variations of the *cut-edges* throughput during runs. Both benchmarks were executed on triangular meshes, although *shift* supports any unstructured mesh.

3.1. Weak scaling

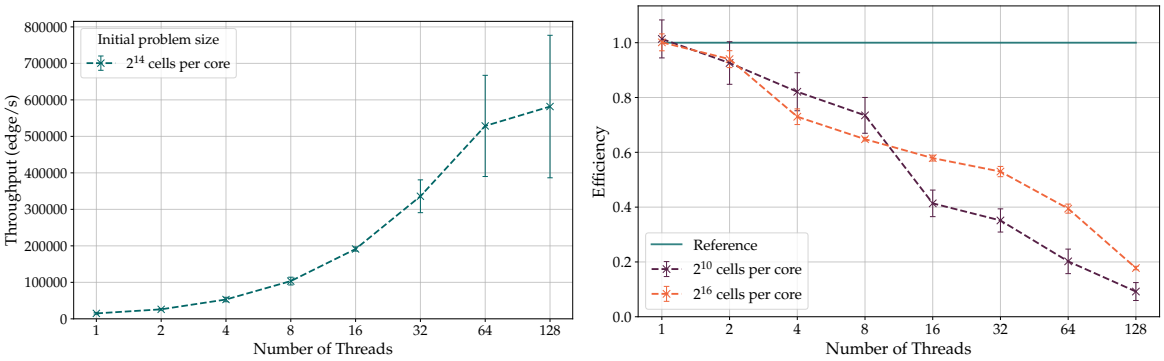
Preliminary results seemed to hint at the importance of problem size in performance. To confirm this, we defined multiple sets of weak scaling runs, each maintaining a different workload per core. Weak scaling results of both benchmarks are shown by Figure 5a and 5b. Ideal efficiency is plotted as reference for the *shift* benchmark.



(a) *cut-edges* throughput (b) *shift* efficiency

Figure 5 – Weak scaling results

While results confirm the tie between performance and per-core workload, the *shift* benchmark highlights a significant efficiency loss. Although not done here, the next step to improve our implementation would be to profile the algorithm and identify the bottleneck(s). It would also be interesting see how our implementation performs against a regular coloring approach, both in scaling and overall processing time: using 128 threads, executing the *shift* operation on a 20 million vertices mesh takes *0.5* second, while executing the *cut-edges* operation on 3 million edges mesh takes less than *5* seconds.



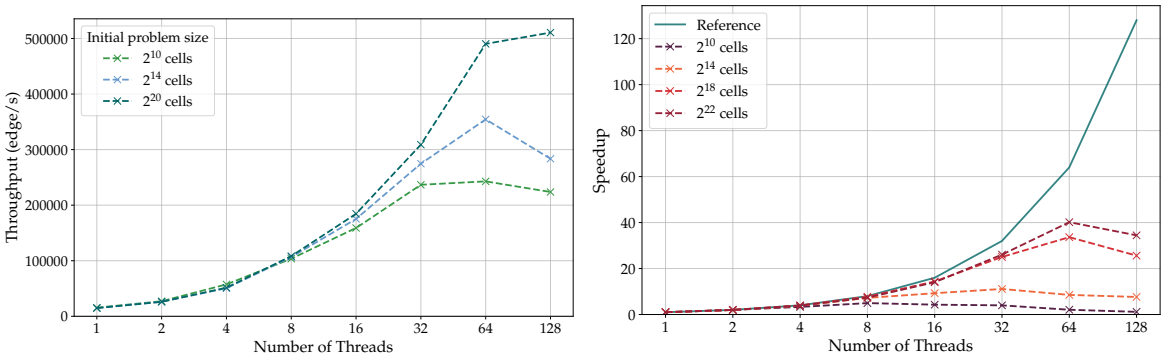
(a) *cut-edges* throughput (b) *shift* throughput

Figure 6 – Standard deviation of weak scaling results

The *shift* benchmark's results seem to indicate that smaller problems not only perform worse, but have much more variation in their execution time. This can be explained with the increased contention that comes with decreasing workload per unit. The *cut-edges* benchmark shows high throughput variation (see Figure 6a) for the same reason: the dynamic workload reaches a maximum, where contention will be minimal, and then decreases when edges have reached the desired target length. In our context, increased contention leads to increased transaction failure, therefore more discarded computations and worse overall execution time.

3.2. Strong scaling

We can confirm previous observations by using a strong scaling study along with a counter of failed transaction. Results are presented in Figure 7 and 8.

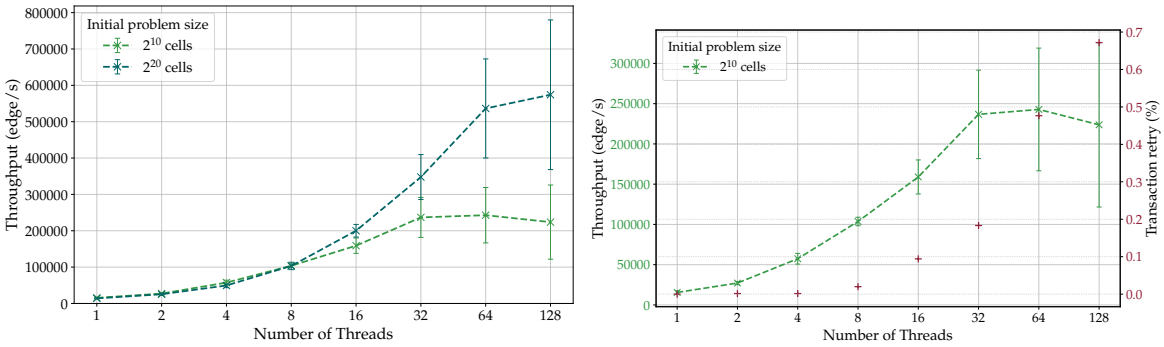


(a) *cut-edges* throughput

(b) *shift* speedup

Figure 7 – Strong scaling results

Unsurprisingly, larger problems perform better. In this case though, the *shift* benchmark shows worse speedup when using 128 threads compared to 64. Aside from problem size, we can speculate that the 2-socket setup of the node holds a role in the performance loss since we did not consider the NUMA profile of the machine when writing the routine, data layout or dispatch logic.



(a) w.r.t. problem sizes

(b) w.r.t. transaction retry rate

Figure 8 – *cut-edges* strong scaling throughput variation

For the *cut-edges* benchmark, throughput evolution reveals that larger problems achieve higher average throughput with more variance. This stems from reaching peak workload per core more quickly, minimizing time spent in high-contention states. Conversely, smaller problems linger longer in low-workload regimes, amplifying contention. The final phase, where workload per core diminishes as edges approach the target length, remains consistent in duration across problem sizes. Figure 8 highlights this and its implication on transaction failures.

4. Conclusion

Our N-maps-based mesh data structure implementation addresses problems specific to parallel remeshing algorithms. Our approach ensures data integrity in concurrent contexts without depending on conflict avoidance.

Relative efficiency and speedup are far from ideal, but this is expected given that our parallelization is intentionally simple, preserving the structure of the original sequential algorithms. For instance, we did not bind threads to cores, optimize dispatch to reduce conflicts, or adjust the data layout to account for cache usage or the machine's NUMA profile.

Near-future work involves profiling our algorithms, particularly the *shift* routine, to identify causes of efficiency loss, as well as conducting comparisons against methods like graph coloring or lock-based synchronization for scalability and performance. These will guide further optimizations and long-term data structure design.

Appendix

A – Library implementation

Our code is available in a public GitHub repository [12]. We used the Rust programming language to implement N-map based data structures, meshing algorithms, and benchmarks. The repository also contains documentation, examples and a simple visual debugging tool to render maps. We implemented the data structure from scratch while leveraging some third-party libraries to handle some specific parts of the code (e.g. `rayon` [13], `vtkio` [14] and `bevy` [15], respectively for parallelization, VTK format support, and the rendering tool). We publish our work as a libraries on Rust's main registry, *crates.io*.

Existing combinatorial maps implementations can be split into two categories: array-based implementations and pointer-based implementations. The former does not actually contain dart objects, but uses dart identifiers to index the storage of beta function values; the latter has dart objects holding their respective beta function images as pointers towards other darts. We choose to use an array-based approach, expecting it to yield better performance in a parallel context. The beta images are stored in a column-major matrix to improve locality.

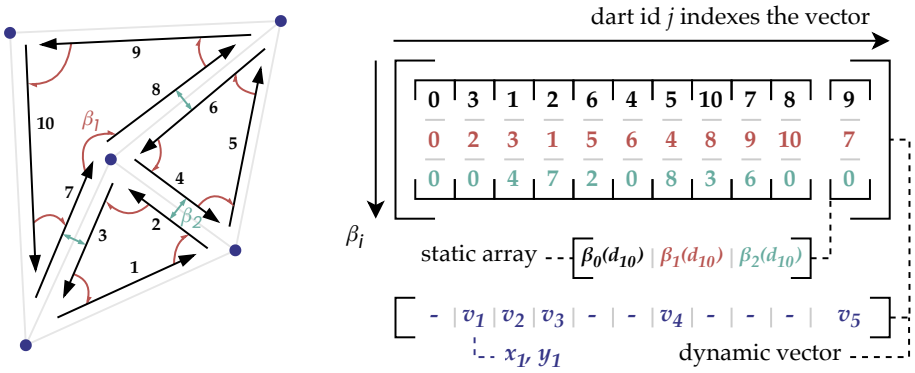


Figure 9 – 2-map representation and its data layout in memory

We implement two distinct data structures for 2-maps and 3-maps, `CMap2` and `CMap3`, to simplify some operator implementations. Map creation is unified through a builder structure, `CMapBuilder`, which also serves as the entrypoint for predefinite maps (e.g. a cartesian grid) and deserialization from two supported formats (VTK Legacy and a custom one).

Data embedding ([1], Chapter 7) is implemented using Rust's traits to model mathematical properties of attributes. Our generic attribute system (implemented in `honeycomb-core/src/attributes/`) allows users to define their own strongly-typed attributes to bind to cells of the mesh.

Our implementation isn't tied to a specific parallelization framework, nor does it enable parallel execution exclusively through provided methods. Our benchmarks use both the `rayon` crate as well as standard threads for parallel dispatch. The latter are also used in the core crate to accelerate deserialization of our custom format.
Synchronization is handled using a fork [16] of the `rust-stm` crate [17]. The fork adds support for transaction cancellations on user-defined errors. This feature simplifies the implementation of nuanced control flow in transactions containing multiple potential causes of failure.

B – Scaling metrics

Ideal execution

We consider our algorithms to be embarrassingly parallel and fully parallelizable. This is an approximation: even if the workload can be split into parallel tasks easily, these cannot be executed in parallel directly as data accesses of different tasks can conflict.

Ideal executions of the algorithms would involve differed execution of conflicting operations, those being detected beforehand (e.g. using coloring). Even if we exclude conflict detection from speedup calculations, the ideal execution may not be embarrassingly parallel due to some operations not being overlappable and/or not enough parallel work being exposed.

The fraction of parallelizable work in a meshing algorithm depends on problem input. Algorithms may even alter that fraction during execution (dynamic workloads). For both of these reasons, we need to work with approximate values of ideal metrics.

Ideal values of measured metrics

For a fixed-size problem, Amdhal's law give us the maximum speedup we can expect from the parallelization of a code. Given the fraction of it that *can* be executed in parallel p and the number of threads used N, the speedup is defined as:

$$S_p(N) = \frac{1}{1 - p + \frac{p}{N}}$$

If we consider our problem to be embarrassingly parallel ($p = 1$), we get the ideal speedup:

$$S_{ref} = \frac{1}{1 - 1 + \frac{1}{N}} = N$$

Gustafson's law gives us an alternative to Amdahl's law which is applicable to studies where problem size scales with available resources. The scaled speedup is defined as:

$$S'_p(N) = 1 + (N - 1) * p$$

For weak scaling run, we choose to compute the efficiency as its ideal value is more intuitive for embarrassingly parallel problems. Efficiency is defined as:

$$E_p(N) = \frac{S'_p(N)}{N}$$

Which, with $p = 1$, gives us the ideal efficiency value:

$$E_{ref} = \frac{N}{N} = 1$$

If a constant workload per core is maintained, we expect the processing time to stay constant, and the fraction $\frac{T_{exec}(n_{threads}=1)}{T_{exec}(n_{threads}=N)}$ will be equal to 1. If the processing time grows (i.e. speedup drops), the efficiency drops below 1.

Bibliography

1. Damiand (G.) et Lienhardt (P.). – *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing*. – A K Peters/CRC Press, septembre 2014.
2. Kraemer (P.), Untereiner (L.), Jund (T.), Thery (S.) et Cazier (D.). – CGoGN: N-dimensional Meshes with Combinatorial Maps. *In : Proceedings of the 22nd International Meshing Roundtable*, éd. par Sarrate (J.) et Staten (M.), pp. 485–503. – Cham, Springer International Publishing, 2014.
3. CGAL 5.6 - Combinatorial Maps: User Manual. – https://doc.cgal.org/latest/Combinatorial_map/.
4. CGoGN Manual. – https://cgogn.github.io/.
5. Rakotoarivelo (H.). – *Contributions au co-design de noyaux irréguliers sur architectures manycore : cas du remaillage anisotrope multi-é chelle en mécanique des fluides numérique.* – PhD Thesis, Université Paris-Saclay, juillet 2018.
6. Marot (C.), Pellerin (J.) et Remacle (J.-F.). – One machine, one minute, three billion tetrahedra. *International Journal for Numerical Methods in Engineering*, vol. 117, n9, mars 2019, pp. 967–990.
7. Shavit (N.) et Touitou (D.). – Software transactional memory. *Distributed Computing*, vol. 10, n2, février 1997, pp. 99–116.
8. O'Sullivan (B.), Stewart (D.) et Goerzen (J.). – *Real World Haskell, Chapter 28*. – O'Reilly Media, novembre 2008.
9. Bihari (B. L.). – Transactional Memory for Unstructured Mesh Simulations. *Journal of Scientific Computing*, vol. 54, n2, février 2013, pp. 311–332.
10. Scott (M. L.), Spear (M. F.), Dalessandro (L.) et Marathe (V. J.). – Delaunay Triangulation with Transactions and Barriers. – In *2007 IEEE 10th International Symposium on Workload Characterization*, pp. 107–113, septembre 2007.
11. Cascaval (C.), Blundell (C.), Michael (M.), Cain (H. W.), Wu (P.), Chiras (S.) et Chatterjee (S.). – Software Transactional Memory: Why Is It Only a Research Toy? The promise of STM may likely be undermined by its overheads and workload applicabilities. *Queue*, vol. 6, n5, septembre 2008, pp. 46–58.
12. LIHPC-Computational-Geometry/honeycomb. – https://github.com/LIHPC-Computational-Geometry/honeycomb, février 2025.
13. Rayon-rs/rayon. – https://github.com/rayon-rs/rayon, mai 2025.
14. Elrnv/vtkio. – https://github.com/elrnv/vtkio, avril 2025.
15. Bevyengine/bevy. – https://github.com/bevyengine/bevy, mai 2025.
16. Imrn99/fast-stm. – https://github.com/imrn99/fast-stm, mai 2025.
17. Marthog/rust-stm. – https://github.com/Marthog/rust-stm, mai 2025.