Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2016) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

25th International Meshing Roundtable (IMR25)

Fine-grained locality-aware parallel scheme for anisotropic mesh adaptation

Hoby Rakotoarivelo[a,b,*], Franck Ledoux[a], Franck Pommereau[b]

[a] *CEA, DAM, DIF, F-91297 Arpajon, France*
[b] *Laboratoire IBISC, Université Paris-Saclay, France*

Abstract

In this paper, we provide a fine-grained parallel scheme for anisotropic mesh adaptation on NUMA[1] architectures.
Data dependencies are expressed by a graph for each kernel, and concurrency is extracted through fine-grained graph coloring.
Tasks are structured into bulk-synchronous steps to avoid data races and to aggregate shared-data accesses.
To ensure performance prediction, time cost and load imbalance are theoretically characterized.
The devised scheme was evaluated on a 4 NUMA node (2-socket) machine, and a mean efficiency of 70% was reached on 32 cores for 3 kernels out of 4. The impact of irregular degree distribution and data layout on scalability is highlighted.

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

Keywords: Irregular applications, Anisotropic Mesh Adaptation, Fine-grained parallel meshing, NUMA-aware algorithms.

1. Introduction

Numerical simulations of complex physical phenomena (such as turbulence, noise propagation) may require billions of mesh elements to achieve a high level of accuracy. Coupled with a parallel numerical scheme, mesh adaptation is a relevant alternative to reduce the computational effort while maintaining the required accuracy. By equidistributing the error of the solution field while modifying the domain discretization, it decreases the required number of degrees of freedom. That said, the whole computational chain must be efficiently parallelized to be applicable, according to Amdahl's law [1]. But mesh adaptation is known as being irregular [2] and difficult to parallelize at a fine granularity level. Indeed, mesh topology is explicitly stored but evolves in an unpredictable way. Data dependencies cannot be resolved statically, involving a poor cache locality due to irregular memory access patterns. On the other hand, manycore architectures have been recently emerging in high performance computing platforms, with an increasing number of cores per node but a decreasing memory/clock rate per core, and a deep cache hierarchy. Furthermore, inter-socket memory accesses have higher latency than intra-socket ones in a NUMA multicore machine.

[1] Non-Uniform Memory Access

* Corresponding author. Tel.: +33 1 77 57 59 12
E-mail address: hoby.rakotoarivelo.ocre@cea.fr

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

Therefore, coarse-grained schemes based on serial kernels are no longer suitable, as pointed out in [3]. The challenge is to devise an efficient fine-grained scheme which exhibit a high degree of concurrency and a high locality to ease memory accesses penalties. The idea is no longer to process larger mesh in a given time frame T, but to shorten T.

Related works. Parallel meshing is an active field of research [4]. Most of up-to-date schemes are coarse-grained ones, and rely on serial kernels, domain decomposition and dynamic cell migration for load balancing. In this case, the parallelization effort mainly focuses on domain interfaces management in order to reduce the synchronization between submeshes, and efficient heuristics for load balancing [5]. Fine-grained schemes have also been developed, but many of them concerns lock-based Delaunay remeshing [6].

In [7], a lock-free scheme for the isotropic case was proposed. Mesh operations were expressed by a graph, and an independent set I is extracted to ensure topological conformity and data updates consistency. The critical point is the extraction of I in the swapping stage because it involves a *distance-2* vicinity for each vertex. Such a choice greatly reduces the degree of concurrency since a *2*-coloring requires more colors.

In [8], an extension to the anisotropic case was given with an additional contraction kernel. It uses a node-to-node and node-to-element data structure and a unique graph was considered for all kernels. To ensure data consistency, adjacency lists updates are deferred to the end of each remeshing iteration in a way that a unique thread k commits the list of a given point p, even if $\mathcal{N}_\mathrm{p}$ is partially updated by other threads. It was designed in a practical way and no theoretical guarantees were provided.

Contributions. We provide a fine-grained scheme for anisotropic remeshing on NUMA architectures based on [7,8]. Data dependencies are expressed by a graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$ and concurrency is extracted through fine-grained graph coloring. Locality is the key-point of our scheme and its originality relies on:

- kernel-specific graphs to increase |V| and the degree of concurrency $|I|$, and to reduce |E|, unlike [8]. (sec. 3.1) That said, no graph is required for refinement and only *distance*-1 edges are considered, unlike [7].
- a locality-aware data layout management in order to ease cache and NUMA effects (sec. 3.2)
- a bulk-synchronous task structuring in order to avoid data races and to aggregate data accesses (sec. 4).
- the use of a bridging model to ensure performance portability, unlike [8] (sec. 5).

2. Serial adaptive scheme

The purpose is to control the discretization of the domain Ω to accurately capture characteristic features of a given solution field $(u_\mathrm{p})_{\mathrm{p}\in\Omega}$ (such as shocks, flows, interfaces motion), while reducing the number of degrees of freedom. It aims at building a discretization M of Ω where the interpolation error $\varepsilon = \|u - \Pi_u\|$ is bounded and equally distributed, with Π_u the interpolate of u on M. It can be expressed as the following optimization problem:

$$\text{find } (u, \mathsf{M}) = \arg\min_{(u_i, \mathsf{M}_i)} \|u_i - \Pi_{u_i}\| \text{ subject to a fixed number of nodes } N \tag{1}$$

This non-linear problem is resolved by applying an iterative procedure involving both a finite element/volume solver and a adaptive remesher. The convergence of the couple mesh-solution is achieved when the error is smaller than a given threshold. The sequence of operations we use is given in Algorithm 1.

2.1. Metric construction

The error of the solution field is closely related to mesh quality: a control of elements size is required to reduce it. For this purpose, one may use a local *a posteriori* error estimate from which a nodewise metric tensor field $(\mathcal{M}_\mathrm{p})_{\mathrm{p}\in\Omega}$ is derived, and will guide the remeshing procedure. In addition, phenomena involved in computational fluid dynamics often admit anisotropic features: the solution field evolves differently depending on the considered direction.
To properly capture them, a direction prescription is thus necessary. For each tensor, its eigenvalues $(\lambda_i)_{i=1,d}$ give the local size prescription $h_i = 1/\sqrt{\lambda_i}$ in the direction of their related eigenvectors $(\mathrm{v}_i)_{i=1,d}$, with d is the dimension of $\mathbf{\Omega}$. Then a gradation procedure may be applied in order to smooth out sudden changes in size requirements [9].

Many error models can be found in the literature, each one being suitable to a specific class of numerical schemes. In our case, we use a generic multi-scale model based on the continuous mesh formalism [10], which gives us a control in L^k norm of the error, for all $k \in \mathbb{N}$. The main benefit of this approach is to give an automatic normalization of the tensor field $(\mathcal{M}_p)_{p\in\Omega}$. Indeed, it finely captures characteristic features with different amplitudes (shock within a flow for instance), without a minimal size requirement. For a target number of points N, the metric tensor of a point p is:

$$\mathcal{M}_p = \underbrace{N\left(\int_\Omega \det(|H_{u|x}|)^{\frac{2k}{2(k+1)}}\,dx\right)^{-1}}_{\text{global normalization}} \underbrace{\det(|H_{u|p}|)^{\frac{-1}{2(k+1)}}}_{\text{local normalization}} \cdot |H_{u|p}|, \text{ with } |H_u| = P \cdot \begin{pmatrix} |\lambda_1| & 0 \\ 0 & |\lambda_2| \end{pmatrix} \cdot P^{-1} \tag{2}$$

The local normalization factor aims to control the point density in large gradient variation regions of Ω, whereas the global normalization factor is required to reach N according to the chosen L^k norm. The metric tensor definition involves the Hessian matrix H_u of u, which is only known at mesh nodes. The hessian field is recovered by means of a double L2-projection, which is accurate and does not tend to smooth the calculated field [11].

2.2. Remeshing

It consists in applying topological and geometric modifications to the mesh with respect to metric field requirements. Formally, the purpose is to build an uniform unit mesh in the Riemannian space induced by $(\mathcal{M}_p)_{p\in\Omega}$, where the length of a segment [ab] and the area of an element K are given by:

$$\ell(a,b) = \int_0^1 ((b-a)\cdot\mathcal{M}_{a+t(b-a)}\cdot(b-a))^{\frac{1}{2}}\,dt, \text{ and } |K| = |K_2|\cdot(\det\mathcal{M}_K)^{\frac{1}{2}}, \text{ with } |K_2| \text{ the euclidean area of K.} \tag{3}$$

It is an iterative procedure consisting of 4 stages:

- *refinement*: it aims at splitting edges whose lengths are larger than $\sqrt{2}$. We use a recursive dissection kernel, since data updates remain local to the element in this case. This aspect is important for its parallelization. Finally, the Steiner point related each marked edge is its midpoint in the Riemannian space.
- *contraction*: it aims at removing edges whose lengths are smaller than $1/\sqrt{2}$. Here we use a vertex collapse kernel, in which the smallest edge (p_1, p_2) incident to a point p_1 is selected at each time. Task ordering is actually important since processing nodes in a breadth-first way rather than a depth-first way avoids the creation of long edges induced by a subsequent collapse of a node p.
- *swapping*: it aims at improving the quality of any element pair (K_1, K_2) by flipping their shared edge if quality is improved i.e $\sum_{i=1}^{2} Q(K_i) < \sum_{i=1}^{2} Q(K'_i)$ and $\min_{i=1,2} Q(K_i) < \min_{i=1,2} Q(K'_i)$, with (K'_1, K'_2) the resulting pair.
- *smoothing*: it aims at relocating each internal point p so that the quality of its stencil $\mathcal{N}_p$ is strictly improved. Several kernel choices are possible, we opted for a smart laplacian where the location of p is: $p + \sum_{p_i\in\mathcal{N}_p} \omega_i(p_i - p)/\ell(p, p_i)$, with $\omega_i \in [0,1]$: relaxation weights such that p remains in the convex hull of $\mathcal{N}_p$

Regarding quality measure, we opted for the one in [12] which takes both element size and shape into account:

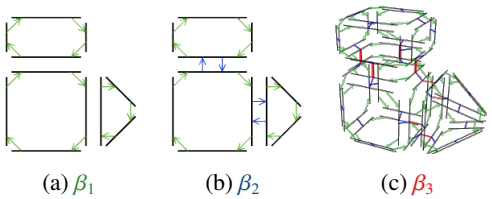
$$\underbrace{12\sqrt{3}|K|\cdot(\partial|K|)^{-2}}_{\text{shape}} \cdot \underbrace{\varphi(\partial|K|/3)}_{\text{size}}, \text{ with } \varphi(1) = 1, \lim_{x\to 0}\varphi(x) = \lim_{x\to\infty}\varphi(x) = 0 \tag{4}$$

- $|K|$ is the Riemannian area of K and $\partial|K|$ is its perimeter according to the tensor interpolated at its vertices.
- φ is defined by $\varphi(x) = [\min(x, \frac{1}{x})\cdot(2 - \min(x, \frac{1}{x})]^3$. It reaches its maximum value 1 for $x = 1$ and decreases smoothly for $x > 1$. Thus $Q(K) = 1$ when K is equilateral with unit sides with respect to $\mathcal{M}$.

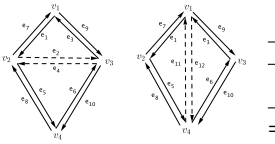
An example of anisotropic mesh adapted to an analytical solution field is given in Fig. 7. No gradation were applied but all scales of the solution field were correctly captured.

input: initial mesh M_0
input: error $\varepsilon_{\max}$, quality q^*, and efficiency τ^* thresholds
  repeat
    solve $(u_\mathrm{p})_{\mathrm{p}\in\Omega}$ on current mesh M ▷ solver
    build a metric tensor field $(\mathcal{M}_\mathrm{p})_{\mathrm{p}\in\mathsf{M}}$ ▷ remesher
    apply gradation on $(\mathcal{M}_\mathrm{p})_{\mathrm{p}\in\mathsf{M}}$ ▷ optionnal
    repeat
      refinement
      contraction
      swapping
      smoothing
    until $q_{\min} \geq \mathrm{q}^*$ and $\tau \geq \tau^*$
  until $\|u - \Pi_u\| \leq \varepsilon_{\max}$
  return (u, M)

Algorithm 1: Adaptive loop



(a) β_1 (b) β_2 (c) β_3



β_2	7	8	9	1
bef.	1	5	3	6
aft.	1	5	3	6

$\Rightarrow$ completely local

(d) flip (e) data updates

Fig. 1: Combinatorial map data structure $\mathrm{D} = (\mathrm{H}, \beta_1, \cdots, \beta_\mathrm{d})$
β_1: permutation on H | $\beta_i, \beta_i \circ \beta_j$: involutions on H, $i > 1, i \neq j$

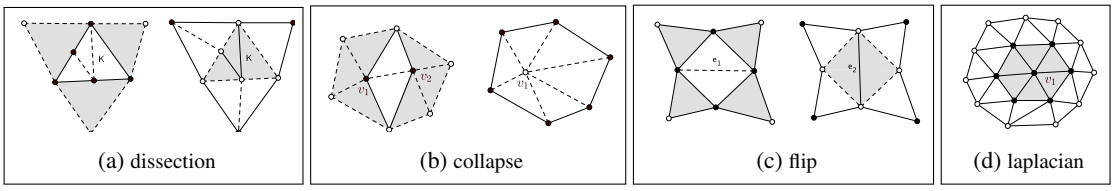
3. Fine-grained parallelization

Issues. Apart from being irregular, the remeshing part is one of those memory-intensive algorithms in which computation is cheap but limited by memory transfers. This aspect is accentuated by irregular memory accesses, induced by mutable data dependencies. It reduces the efficiency of prefetching techniques for memory latency hiding [13]. On the other hand, this level of granularity involves frequent thread synchronization and idle times. To reach scalability, the challenge is to increase task locality, especially in a NUMA context, and reduce synchronization count while keeping the execution safe.

Philosophy of the method. Locality is the key-point of our devised scheme. Data dependencies are expressed by an undirected graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$, from which a task partition $\mathbb{P}$ is extracted by a fine-grained parallel graph coloring. Task granularity is refined to increase the degree of concurrency $|I|$, whereas data dependencies is reduced to increase locality. Thus, graphs are chosen such that |V| is increased and |E| is reduced. Each stage is structured into bulk-synchronous steps in order to avoid conflictual data load/store, and to aggregate shared-data accesses for memory latency hiding. By the way, reordering instructions allows us to exhibit theoretical prediction on execution time and load imbalance.

3.1. Task extraction

To ensure correctness, two issues must be addressed: topological conformity and data consistency. Overlapping operations may invalid the mesh (edge cross, holes), whereas cell data may be corrupted by data races. In our context, a remeshing task is defined by a kernel and a related dataset. Task graphs are built from data dependencies of each kernel, and are chosen such that |V| is increased and |E| is reduced. Indeed, we aim at reducing the degree Δ of each graph, as well as the deviation σ of their degree distribution to reduce load imbalance. On the other hand, the number of colors n_c required to build $\mathbb{P}$ (and $I \in \mathbb{P}$) is bounded by $\Delta + 1$, and must be reduced since $|I|$ decrease linearly to n_c. Data involved by each task are described in Fig. 2. Related graphs are given in Fig. 3 and recapitulated in Table 1.



(a) dissection (b) collapse (c) flip (d) laplacian

Fig. 2: Data involved by each task and propagation

Table 1: Data dependencies and related graphs

task	depend.	unit cost	graph	Δ	χ
dissection	dynamic	$O(1)$	none	N/A	N/A
collapse	dynamic	$O(\lvert\mathcal{N}_\mathsf{p}\rvert)$	primal	var.	6
flip	dynamic	$O(1)$	dual edge	4	6
laplacian	static	$O(\lvert\mathcal{N}_\mathsf{p}\rvert)$	primal	var.	6

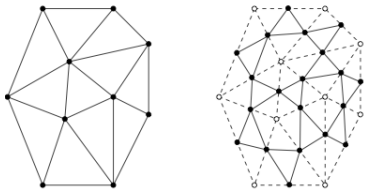
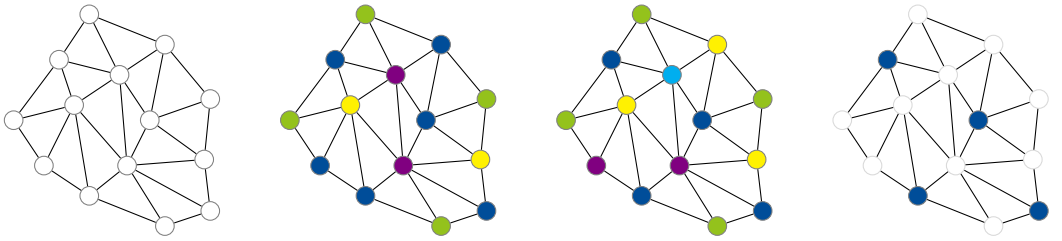


Fig. 3: Related graphs: primal (left) and dual edge (right)

- *dissection*: it involves the vicinity of each element, however this dependency may be avoided by carefully choosing the mesh data structure. No graph is required here, unlike [7].
- *collapse*: it involves the vicinity (stencil) of each node. $\mathbb{G} = (\mathrm{V}, \mathrm{E})$ is built from the mesh primal graph, where V is a set of marked nodes, and E is a set of pairs of V corresponding to a mesh edge. Here $\mathbb{G}$ is planar and we have a bound on the chromatic number $\chi \leq 6$, as stated in [14]. Hence a small number of colors is expected for the extraction of a partition $\mathbb{P}$, despite irregular degree distribution induced by anisotropy.
- *flip*: it involves the shell of the edge (i.e the two elements sharing it). $\mathbb{G}$ is built from the mesh dual edge graph. As each shell contains exactly 4 edges in 2D, we have $\Delta = 4$ (whereas $\Delta = 12$ in [7] and $\Delta \approx 6$ in [8]).
- *laplacian*: it involves the stencil of each internal node. $\mathbb{G}$ is built from the primal graph deprived of its boundary nodes. As the topology remains unchanged, the same partition $\mathbb{P}$ is kept throughout the stage, and task propagation is ensured by processing $\mathrm{V}_i \in \mathbb{P}$ at iteration i.

The devised scheme relies on an efficient independent task extraction which is NP-hard. Related heuristics are not trivially parallelizable due to intrinsic dependencies between iterations [15]. Since the extraction of I is only a preliminary step of a given remeshing stage, its cost must remain negligible. However, the quality of the solution must remain acceptable in order to increase the degree of concurrency $|I|$. Given these constraints, we opted for a speculative fine-grained graph coloring [16]. It gives a good tradeoff between accuracy and performance.



(a) task graph (b) pseudo-coloring (c) defective nodes fixes (d) independent tasks

Fig. 4: Fine-grained speculative graph coloring heuristic

Table 2: Evaluation of the graph coloring heuristic on RMAT instances [17], $|\mathsf{V}| = 16 \cdot 10^6, |\mathsf{E}| = 128 \cdot 10^6$.
Runs are performed on an 2-socket Intel Haswell with a NUMA-aware memory allocation and a static thread binding (1/core).

graph	Δ	$\bar{\Delta}$	σ^2	CPU time (s)				defective vertices				ratio indep. (%)			
				p=1	p=4	p=16	p=32	p=1	p=4	p=16	p=32	p=1	p=4	p=16	p=32
RMAT-ER	51	9.16	21	3.56	1.87	0.73	0.48	0	5	20	46	34.6	31.5	32.9	32.8
RMAT-G	606	3.13	22	1.09	0.63	0.42	0.31	0	1	17	38	60.9	63.9	62.3	62.1
RMAT-B	5398	1.72	123	0.51	0.35	0.28	0.25	0	3	27	31	85.1	88.3	89.4	89.5

The idea is to perform a pseudo-coloring of V using first-first heuristic [18] without worrying about data races, and to handle defective vertices in a separate stage. In the first stage, each thread maintain a local mapping of forbidden colors for each $v \in \mathrm{V}$, because they are already assigned to a $v_i \in \mathcal{N}_\mathsf{v}$. In the second stage, threads recheck colors of a subset of marked vertices R, which will be colored again in the next iteration. If conflict happens, only one of the 2 defective vertices is recolored, which are discriminated by their ID. The algorithm includes 2 synchronization points.

The heuristic was evaluated on 3 instances of the *recursive matrix* graph model (RMAT) [17], to assess its applicability in our context. Indeed, the primal graph (involved in contraction and smoothing stages) has an irregular degree distribution because of anisotropy. The graph model allows to generate instances with degree distribution parameters (see [17] for details). Results are shown in table 2. Less than 4 seconds is required to process $16 \cdot 10^6$ vertices and the number of defective vertices remains negligible compared to |V|. The ratio of independent vertices is not altered by parallelism, but depends on the degree distribution.

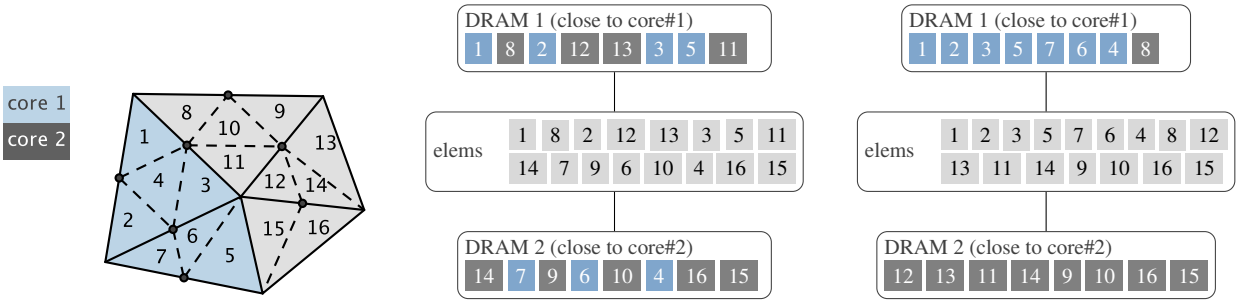
3.2. Locality enhancement

Issues. In our context, data layout must be considered to improve cache locality despite irregular memory accesses. However the initial memory layout is hard to preserve because cell data dependencies evolve in a unpredictable way. On the other hand, index reordering schemes (Hilbert curves, matrix renumbering) are still expensive and hard to parallelize in a fine-grained way. Therefore, the challenge is to reduce data updates on one hand, and keep neighboring cells as close as possible in memory on the other hand, subject to these constraints.

Data layout and synchronization. Data are stored in flat arrays in order to increase spatial locality due to memory addresses contiguity. No reallocation is performed within a remeshing stage but a parallel mesh compression stage is done after a contraction/refinement. To reduce remote accesses in a NUMA context, memory cells are first-touched by each thread in a round-robin fashion for each shared tasklist. Therefore, memory cells are allocated on the closest DRAM of the core where a given thread is statically bound. Data stores are synchronized in a way that newly inserted neighboring mesh cells are kept as close as possible in memory (see Fig. 5). Indeed, mesh cell index offsets n_α, n_β, and n_γ are precalculated according to the number of long edges α_j of each element K_j during a refinement stage:

$$n_\alpha = \sum_{j=1}^{|L|} \alpha_j, \quad n_\beta = \sum_{j=1}^{|L|} \beta_j \quad \text{and } n_\gamma = \sum_{j=1}^{|L|} \gamma_j, \text{ where } \beta_j = \begin{cases} 4 & \text{if } \alpha_j = 1 \\ 8 & \text{if } \alpha_j = 2 \\ 12 & \text{otherwise} \end{cases}, \text{ and } \gamma_j = \begin{cases} 2 & \text{if } \alpha_j = 1 \\ 3 & \text{if } \alpha_j = 2 \\ 4 & \text{otherwise} \end{cases} \tag{5}$$

A lock-free synchronization scheme is used for shared tasklist updates: a reduction is performed on index offsets (k_i) for a shared tasklist $\mathbb{L}$, in order to find the current index range where a given thread t_i may store its private data. For that, a atomic capture mechanism inspired by [8] is used: the size n_L of $\mathbb{L}$ is incremented atomically while its old value is cached in k_i. Thus, t_i knows its index range $[\mathsf{k}_i, \mathsf{k}_i + n_\mathsf{L}]$ and can resume its local computation or copy his local data. Notice that offsets could be retrieved by means of prefix sum, but it would require $\log(p)$ synchronization barriers.



(a) mesh cells to be inserted (b) asynchronous: 7 remote stores (c) synchronized: 1 remote store

Fig. 5: Data insertion scheme during a refinement stage, in a asynchronous way (b) and in a synchronized / NUMA-aware way (c)

Data structure. A combinatorial map data structure is used such that updates remain local to the considered patch [19] (see Fig. 1). Each dart (or halfedge) stores the indices of its vertices, its next, its twin, and the element containing it. In addition, each node stores the index of an outgoing dart, idem for elements. The stencil of each node can be easily found via a circulator. Boundary darts are linked so that the whole boundary may be retrieved via references traversal.

4. Steps structuring

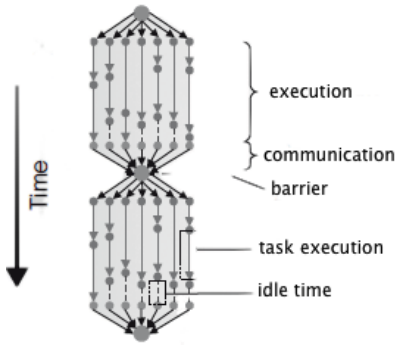


Fig. 6: A bulk-synchronous computation. It consists of a local computation, a communication step followed by a synchronization barrier.

Remeshing stages are structured into bulk-synchronous steps for 3 reasons:

- *avoid data races*: threads use local copies of shared-data, and updates are made in a synchronized way at the end of each step, using the mechanism described in section 3.2. Thus, conflictual data load/stores are avoided.
- *hide memory latency*: shared-data accesses are aggregated in a single communication substep, and may be performed in a pipelined way.
- *ensure performance prediction*: the time cost of each step may be finely estimated through bridging models for a particular machine, given a set of parameters (unit task costs, bandwidth, latency, caches etc) [20–22].

(see Fig. 6)

Refinement. Each iteration consist of 4 steps: (1) element marking, (2) steiner point and offsets calculation, (3) element process, (4) twin darts repairing. In addition to the mesh, threads share 2 arrays: $\mathbb{L}$ storing the indices of marked darts, and $\mathbb{S}$ for the Steiner point related to each $e \in \mathbb{L}$. Related index offsets n_{L} and n_{S} are also maintained.

If an edge $e_{\mathsf{k}} \in \mathsf{K}$ is too long, then the element K is marked and the index k is added to a local list L. A reduction on n_{L} is performed by each thread in order to know the current offset of $\mathbb{L}$ where it may copy the content of L. The same applies to n_{S} with $\mathbb{S}$. Here, only 3 reductions per thread is necessary to update them in the mesh data structure. Finally when a dart $e : (\mathrm{a}, \mathrm{b}) \in \mathbb{L}$ is split into (e_1, e_2), the stored indices of (a, b) are replaced by those of (e_1, e_2). Thus no additional shared array is required to perform the last step (see Algorithm 2).

Contraction. Each iteration consist of 4 steps: (1) stencil analysis and graph construction, (2) graph coloring $\mathbb{P}$, (3) process of $I \in \mathbb{P}$, (4) boundary edges repairing. Threads share a graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$ and 3 additional arrays: S indexing the dart to be collapsed for each point, $\mathbb{L}$ for active nodes indices, and C storing the colors of each point p. An offset n_{L} is also maintained for the latter.

First, the stencil $\mathcal{N}_i$ of each p_i is retrieved. Then edges $e \in \mathcal{N}_i$ are sorted according to their lengths, such that $\mathrm{S}[i] = \arg\min_{\mathsf{k}}\{\ell(e_{\mathsf{k}}) \mid e_{\mathsf{k}} \in \mathcal{N}_i \wedge \ell(e_{\mathsf{k}}) \leq 1/\sqrt{2}\}$. Then for each point p_{k}, if $\mathrm{S}[\mathsf{k}]$ is a valid index then k is added to a local list L, and a reduction is performed on n_{L} in order to find the current offset of $\mathbb{L}$ from where each thread may copy the content of their respective list L. Afterward $\mathbb{G}$ is built by storing each $\mathrm{p}_{\mathsf{k}} \in \mathbb{L}$ in V and its stencil $\mathcal{N}_{\mathsf{k}}$ in E. Then the partition $\mathbb{P}$ is extracted from $\mathbb{G}$, and I from $\mathbb{P}$ by comparing $|\mathrm{P}_c|_{c \in [1, n_{\mathrm{c}}]}$. Each point $\mathrm{p} \in I$ is then processed. Finally, each boundary dart is fixed by updating its *next* reference (see Algorithm 3).

Swapping. Here we have 3 steps per iteration: (1) edge filtering and graph construction (2) graph coloring $\mathbb{P}$, (3) process of $I \in \mathbb{P}$. Threads share a graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$ and an array $\mathbb{L}$ referencing darts which need to be processed. An index offset n_{L} is also maintained.

If $\mathrm{Q}_{\mathcal{M}}(\mathsf{K}) \leq \mathrm{q}_{\mathsf{min}}$ then the index of each $e_{\mathsf{k}} \in \mathsf{K}$ is added to a local list L. A reduction on n_{L} is performed in order to find the current offset of $\mathbb{L}$ from which each thread may copy the content of its list L. Afterward $\mathbb{G}$ is built by storing each $e_{\mathsf{k}} \in \mathbb{L}$ in V and indices of the shell of e_{k} is retrieved and stored in E. Then the partition $\mathbb{P}$ and $I \in \mathbb{P}$ are extracted. Each $e \in I$ is then processed (see Algorithm 5).

Smoothing. Unlike the 3 others, this stage consist of a preprocessing step and 1 step per iteration. The preprocessing consists of: (a) quality analysis, (b) graph construction, (c) graph coloring. Since mesh topology remains unchanged, many steps are deported in the preprocessing. Afterward, each subset $(\mathrm{V}_i)_{i \in [1, n_{\mathrm{c}}]} \in \mathbb{P}$ is processed (see Algorithm 4).

shared : mesh M, tasklist $\mathbb{L}$, offsets n_{L}, n_α, n_β and n_γ.
private: tasklist L, offsets: $\mathsf{k}, \mathsf{k}_\beta, \mathsf{k}_\gamma, \mathsf{off}_\alpha, \mathsf{off}_\gamma$
repeat
  $\mathbb{L} \leftarrow \emptyset$, $n_{\mathrm{L}} \leftarrow 0$.

---

  for element K **do in parallel** **no wait** ▷ STEP 1
    if there is a short edge $e_i \in \mathsf{K}$ then mark K.
    add $\min[i, \mathrm{twin}_i]$ to L
  atomic⟨ $\mathsf{k} \leftarrow n_{\mathrm{L}}$ and $n_{\mathrm{L}} \leftarrow \mathsf{k} + |L|$ ⟩ ▷ communication
  copy $\mathsf{K}_i \in L$ to $\mathbb{L}$ at index $\mathsf{k} + i$ and reset L and k.
  barrier

---

  for $i \in [1, n_{\mathrm{L}}]$ **do in parallel** **no wait** ▷ STEP 2
    calculate the steiner point of $\mathbb{L}[i]$ and store to L
  atomic⟨ $\mathsf{k} \leftarrow n_\alpha$ and $n_\alpha \leftarrow \mathsf{k} + |L|$ ⟩ ▷ communication
  copy each $\mathsf{p}_i \in L$ into M at index $\mathsf{k} + i$ and reset L.
  barrier

  $\mathsf{off}_t \leftarrow \mathsf{off}_h \leftarrow \mathsf{k}_h \leftarrow \mathsf{k}_t \leftarrow 0$ ▷ STEP 3
  for marked element K_i **do in parallel** **no wait**
    $\mathsf{k}_\beta \leftarrow \mathsf{k}_\beta + \beta_i$ and $\mathsf{k}_\gamma \leftarrow \mathsf{k}_\gamma + \gamma_i$ ▷ see equation 8
    add i into L
  reserve $(\mathsf{k}_\beta + \mathsf{k}_\gamma)$ memory chunks in M. ▷ communication
  atomic⟨ $\mathsf{off}_\beta \leftarrow n_\beta$ and $n_\beta \leftarrow \mathsf{off}_\beta + \mathsf{k}_\beta$ ⟩
  atomic⟨ $\mathsf{off}_\gamma \leftarrow n_\gamma$ and $n_\gamma \leftarrow \mathsf{off}_\gamma + \mathsf{k}_\gamma$ ⟩
  for $i \in [1, \mathsf{k}_t]$ **do**
    refine $L[i]$, mark new cells, update off_β and off_γ.
  barrier

---

  for marked element K **do in parallel** ▷ STEP 4
    repair twin index of e, for each dart $e \in \mathsf{K}$
  barrier
until $\mathbb{L} = \emptyset$

Algorithm 2: bulk-synchronous refinement

shared : mesh M, graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$, partition $\mathbb{P}$.

---

for element K **do in parallel** ▷ preprocess 1
  calculate and cache Q(K) in mesh.
barrier
for node $\mathsf{p}_i \in \Omega - \partial\Omega$ **do in parallel** ▷ preprocess 2
  $\mathrm{V}[i] \leftarrow \mathsf{p}_i$
  retrieve $\mathcal{N}_i$ and store to E_i.
barrier
extract partition $\mathbb{P}$ from $\mathbb{G}$ in parallel ▷ preprocess 3

---

repeat
  for $i \in [1, n_{\mathrm{c}}]$ **do**
    for point $\mathsf{p}_{\mathsf{k}} \in \mathrm{P}_i$ **do in parallel** ▷ STEP i
      attempt to relocate p_{k} regarding $\mathcal{N}_{\mathsf{k}}$
    barrier
until max iteration is reached

Algorithm 4: bulk-synchronous smoothing

shared : mesh M, graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$
shared : partition $\mathbb{P}$, tasklists $\mathbb{L}$ and I, offset n_{L}.
private: tasklist L, offset k
repeat
  $\mathbb{L} \leftarrow \emptyset$, $n_{\mathrm{L}} \leftarrow 0$, reset $\mathbb{G}$

---

  for point $\mathsf{p}_i \in \mathsf{M}$ **do in parallel** **no wait** ▷ STEP 1
    retrieve and cache $\mathcal{N}_i$, identify $\mathbb{S}[i]$ accordingly.
    if $\mathbb{S}[i]$ is valid **then**
      add i to L
  atomic⟨ $\mathsf{k} \leftarrow n_{\mathrm{L}}$ and $n_{\mathrm{L}} \leftarrow \mathsf{k} + |L|$ ⟩ ▷ communication
  copy each $\mathsf{p}_i \in L$ into V at index $\mathsf{k} + i$ and reset L.
  for $i \in [\mathsf{k}, \mathsf{k} + |L|]$ **do**
    $\mathrm{E}[i] \leftarrow \mathcal{N}[\mathrm{V}[i]]$
  barrier

---

  extract partition $\mathbb{P}$ from $\mathbb{G}$ in parallel, and $I \in \mathbb{P}$ ▷ STEP 2
  barrier

---

  for i from 1 to $|I|$ **do in parallel** ▷ STEP 3
    merge $I[i]$ to $\mathbb{S}[i]$
  barrier

---

  for dart $e \in \partial\Omega$ **do in parallel** ▷ STEP 4
    repair *next* reference of e using a circulator.
  barrier
until $\mathbb{L} = \emptyset$

Algorithm 3: bulk-synchronous contraction

shared : mesh M, graph $\mathbb{G} = (\mathrm{V}, \mathrm{E})$, tasklist $\mathbb{L}$, offset n_{L}.
private: tasklist L, offset k
repeat
  $\mathbb{L} \leftarrow \emptyset$, $n_{\mathrm{L}} \leftarrow 0$, reset $\mathbb{G}$

---

  for element K **do in parallel** **no wait** ▷ STEP 1
    calculate and cache $\mathrm{Q}_{\mathcal{M}}(\mathsf{K})$ in M.
    if $\mathrm{Q}_{\mathcal{M}}(\mathsf{K}) \leq \mathsf{q}_{\mathsf{min}}$ **then**
      add each $\min\{e_i, twin(e_i)\} \in \mathsf{K}$ in L
  atomic⟨ $\mathsf{k} \leftarrow n_{\mathrm{L}}$ and $n_{\mathrm{L}} \leftarrow \mathsf{k} + |L|$ ⟩ ▷ communication
  copy each $e_i \in L$ into V at index $\mathsf{k} + i$ and reset L.
  for $i \in [\mathsf{k}, \mathsf{k} + |L|]$ **do**
    retrieve the shell of $\mathrm{V}[i]$ and store to $\mathrm{E}[i]$
  barrier

---

  extract partition $\mathbb{P}$ from $\mathbb{G}$ in parallel, and $I \in \mathbb{P}$ ▷ STEP 2
  barrier

---

  for $1 \leq i \leq |I|$ **do in parallel** ▷ STEP 3
    attempt to flip $I[i]$
  barrier
until $\mathbb{L} = \emptyset$

Algorithm 5: bulk-synchronous swapping

5. Performance prediction

The time cost of each remeshing stage is estimated with the *queuing shared memory* bridging model [22,23]. It consists of p cores, each with its own private memory, and communicating through load/stores within n memory cells in a shared-memory or distributed shared-memory (DSM). Any cell $j \leq n$ may be simultaneously accessed, but there is a cost κ_j for such contention. The *gap* between local instruction rate and communication rate is given by a parameter $g \geq 1$. In this framework, the time cost of a bulk-synchronous remeshing stage is given by:

$$t = \sum_{s=1}^{n_{\text{steps}}} \Big(\underbrace{\max_{i=1,p} c_i}_{\text{local comput.}} + \underbrace{g \cdot \max_{i=1,p}[r_i, w_i]}_{\text{communication}} + \underbrace{\max_{j=1,n} \kappa_j}_{\text{contentions}} \Big) \tag{6}$$

Finer prediction can be obtained by providing unit task costs per step (cycles, bytes), memory bandwidth (GB/s). The mean time complexity and load imbalance of each stage are given in table 3, and detailed below.

Table 3: Decomposition and complexity of each stage. $\bar{\Delta}$: mean graph degree, $n_{\mathsf{p}}, n_{\mathsf{h}}, n_{\mathsf{k}}$: number of nodes, darts and elements $\mathbb{k}$: number of iterations for graph coloring, $\sigma_{\max} = (\max_{i=1,p} \bar{\Delta}_{s,i}) - (\frac{1}{p}\sum_{i=1}^{p} \bar{\Delta}_{s,i})$ with $\bar{\Delta}_{s,i}$: mean graph degree at step s on core i.

stage	nb	bulk-synchronous steps				static task partitioning		
		1	2	3	4	time cost	imbalance	κ
graph coloring	2	process	detect	-	-	$\Omega(g\bar{\Delta}n/p)$	$\Omega(\sigma_{\max})$	2
refinement	4	filter	steiner	process	repair	$\Omega((1+g)n_{\mathsf{K}}/p)$	0	8
contraction	4	stencil	graph	process	repair	$\Omega((1+g)(\mathbb{k}+\bar{\Delta})n_{\mathsf{p}}/p)$	$\Omega(\Sigma_{s=1}^{4}\sigma_{\max,s})$	$2\mathbb{k}+2$
swapping	4	filter	graph	process	-	$\Omega((1+g)(2\mathbb{k}+4)n_{\mathsf{h}}/p)$	0	$2\mathbb{k}$
smoothing	$1+n_c$	preproc	process	process	process	$\Omega((1+g)\bar{\Delta}n_{\mathsf{p}}/p)$	$\Omega(\sigma_{\max})$	$2\mathbb{k}$

Proposition 1 (Refinement)**.** *Let n be the number of mesh elements. The time cost of a refinement iteration is in $\Omega((1+g)\frac{n}{p})$ and no substantial load imbalance is expected with a static task partitioning.*

Proof. For each core i and at a given iteration, we have:

- step 1: let $n_{1,i}$ be the number of elements assigned to the core i. Hence $r_i = n_{1,i}$ are read.
  For each element, at most 3 lengths computations are performed and 1 local value is added to local list, thus $c_i = 4n_{1,i}$. A reduction on $n_{\mathbb{L}}$ is performed involving 1 read/1 write per core, thus $\kappa = 2$.
  Finally, at most $n_i \leq \sum_{\mathsf{k}=1}^{p} n_{1,\mathsf{k}}$ values are written into $\mathbb{L}$ so $w_i = \Omega(n_{1,i})$.
- step 2: let $n_{2,i}$ be the number of long darts assigned to the core i. Thus $r_i = n_{2,i}$.
  Here, r_i steiner point calculations are done, and r_i values are stored locally, so $c_i = 2n_{2,i}$.
  A reduction on n_α is performed, thus $\kappa = 2$. Finally, $n_{\mathsf{p}} = r_i$ values are written to mesh, so $w_i = r_i$.
- step 3: let $n_{3,i}$ be the number of marked elements assigned to the core i. Thus $r_i = n_{3,i}$.
  For each element, 2 offset calculations are done, 1 value is stored locally and 1 refinement is done.
  Hence $c_i = 4n_{3,i}$. Then 2 reductions on n_β and n_γ are done involving 1 read/1 write each other, so $\kappa = 4$.
  Finally $(n_\beta + n_\gamma)$ values are written by all cores, with $n_\beta \leq 3n_\gamma$ and $n_\gamma \leq 4n$, making a total of $n(12+4)$.
  Thus $w_i = \Omega(16n_{3,i})$.
- step 4: let $n_{i,4}$ be the number of new elements assigned to the core i. Hence $r_i = n_{i,4}$.
  For each element, 3 dart updates are performed, so $c_i = 3n_{i,4}$.
  In addition, 3 values per element are written in mesh, thus $w_i = 3n_{i,4}$. No reduction is performed so $\kappa = 0$.

Therefore we have: $t = \sum_{s=1}^{4} \max_{i=1,p}[\Omega(n_{s,i})] + g\sum_{s=1}^{4} \max_{i=1,p}[\Omega(n_{s,i})] + 8$ for any scheduling policy. For a static task partitioning, we have: $n_{s,i} = \frac{n_s}{p} = \Omega(\frac{n_{\mathsf{K}}}{p})$ thus $t = \Omega(\sum_{i=1}^{4}(1+g)\frac{n_s}{p} + 8) = \Omega((1+g)\frac{n_{\mathsf{K}}}{p})$, with n_{K} the nb of elements. Since tasks are equi-distributed and have a uniform unit cost $\Omega(g)$, the load imbalance of a step s is $\iota_s = 0$ □

Proposition 2 (Graph coloring)**.** *Let n_i be the number of vertices assigned to core i, $\bar{\Delta}$ the mean degree of $\mathbb{G}$. For each core i, the local instructions, shared-data accesses and contention count of a graph coloring iteration are: $r_i = \Omega(\bar{\Delta}n_i)$, $w_i = \Omega(2n_i)$, $c_i = n_i$, and $\kappa = 2$.*

Proof. Let R be the shared array storing the indices of conflictual vertices, and n_R its related offset. Then we have:

- pseudo-coloring: for each vertex v, the color of each $v_k \in \mathcal{N}_v$ is retrieved locally and the color of v is updated accordingly. Thus $r_i = \Omega(\bar{\Delta} n_i)$ and $w_i = n_i$.
- defective vertices detection: the colors of each $v_k \in \mathcal{N}_v$ is re-checked for each vertex v. If a same color was assigned to v and v_k, then v_k is added to a local list R_i. Thus $r_i = \Omega(2\bar{\Delta} n_i)$ at this point, and $c_i = n_i$.
  A reduction on n_R is performed, thus $\kappa = 2$. Then the contents of R_i are copied to R, so $w_i = \Omega(2n_i)$.

At this point, we have $r_i = \Omega(\bar{\Delta} n_i)$, $w_i = \Omega(2n_i)$, $c_i = n_i$ and $\kappa = 2$.
The whole procedure is repeated by processing R instead of V, and terminates when $n_R = |R| = 0$. □

Proposition 3 (Contraction)**.** *Let n be the number of nodes, $\bar{\Delta}_i$ the mean degree of the graph on each core i, $\Bbbk$ the number of iterations for graph coloring, and $\sigma_{\max}$ the maximum deviation of $(\bar{\Delta}_i)$ on all cores. The cost of a contraction iter. is in $\Omega((1+g)(\bar{\Delta}+\Bbbk)\frac{n}{p})$, and the load imbalance of each step is in $\Omega(\sigma_{\max})$ with a static task partitioning.*

Proof. Let $(n_{s,i})_{i=1,p}$ be the number of nodes assigned to core i at step s, and $(\bar{\Delta}_{s,i})_{i=1,p}$ the mean degree of $\mathbb{G}$ on i.

- step 1: the stencil $\mathcal{N}_p$ of each point p is retrieved in local memory, thus $r_i = \Omega(\bar{\Delta}_{1,i} \cdot n_{1,i})$.
  Then each $p_i \in \mathcal{N}_p$ is evaluated, and at most $n_{1,i}$ point indices are stored locally, so $c_i = (\bar{\Delta}_{1,i} + 1)n_{1,i}$.
  Then $n_{1,i}$ cells of S are written, so $w_i = n_{1,i}$ at this point. A reduction on n_L is performed, thus $\kappa = 2$.
  Afterward the construction of $\mathbb{G}$ involves at most $n_{1,i}$ writes into V and $(\bar{\Delta}n)_{1,i}$ writes into E, thus $w_i = (\bar{\Delta}n)_{1,i}$.
- step 2: it refers to graph coloring (see Proposition 2). Thus we have $r_i = \Omega(\bar{\Delta}\Bbbk n_{2,i})$, $w_i = \Omega(2\Bbbk n_{2,i})$, $c_i = \Bbbk n_{2,i}$ and $\kappa = 2\Bbbk$, with $\Bbbk$ the number of iterations for coloring $\mathbb{G}$.
- step 3: the stencil $\mathcal{N}_p$ of each point p is retrieved locally, thus $r_i = (\bar{\Delta}n)_{3,i}$.
  The point merge is performed by replacing each stored reference of p_k in $\mathcal{N}_k$ by S[k] and by deleting 2 elements. Thus $w_i = c_i = (\bar{\Delta}n)_{3,i}$ and no reduction is performed, so $\kappa = 0$.
- step 4. Again, the stencil $\mathcal{N}_k$ of each point p_k is retrieved locally, thus $r_i = \Omega(\bar{\Delta}_{4,i} n_{4,i})$. Then each edge $e \in \mathcal{N}_k$ is evaluated so $c_i = r_i$. Afterward one dart is updated so $w_i = 1$. No reduction is performed, so $\kappa = 0$.

Thus we have: $t = \sum_{s=1}^{4}[\max_{i=1,p}\{\Omega(n_{s,i}(\bar{\Delta}_{s,i}+\Bbbk))\} + g\max_{i=1,p}\{\Omega(n_{s,i}(\bar{\Delta}_{s,i}+\Bbbk)\})] + 2(\Bbbk+1)$ for any scheduling policy.
For a static task partitioning, we have: $n_{s,i} = \Omega(\frac{n_p}{p})$, thus $t = \Omega((1+g)\frac{n_p}{p}\sum_{s=1}^{4}(\frac{1}{p}\sum_{i=1}^{p}\bar{\Delta}_{s,i} + \Bbbk) + 2(\Bbbk+1))$ with n_p the number of nodes. Therefore we have: $t = \Omega((1+g)(\bar{\Delta}+\Bbbk)\frac{n_p}{p})$, with $\bar{\Delta} = \frac{1}{4p}\sum_{s=1}^{4}\sum_{i=1}^{p}\bar{\Delta}_{s,i}$. For a step s, the workload ω_i of a core i is in $\Omega((1+g)(\bar{\Delta}_{s,i}+\Bbbk))$, and as g and $\Bbbk$ are fixed, it varies according to $\bar{\Delta}_{s,i}$. Thus the load imbalance is $\iota_s = (\max_{i=1,p}\omega_i) - (\frac{1}{p}\sum_{i=1}^{p}\omega_i) = \Omega(\sigma_{\max})$, with $\sigma_{\max} = (\max_{i=1,p}\bar{\Delta}_{s,i}) - (\frac{1}{p}\sum_{i=1}^{p}\bar{\Delta}_{s,i})$ the max. deviation of $\bar{\Delta}_{s,i}$. □

Proposition 4 (Swapping)**.** *Let n be the number of darts, and $\Bbbk$ the number of iterations for graph coloring. The time cost of a swapping iteration is in $\Omega((1+g)(2\Bbbk+4)\frac{n}{p})$, and no substantial load imbalance is expected with a static task partitioning.*

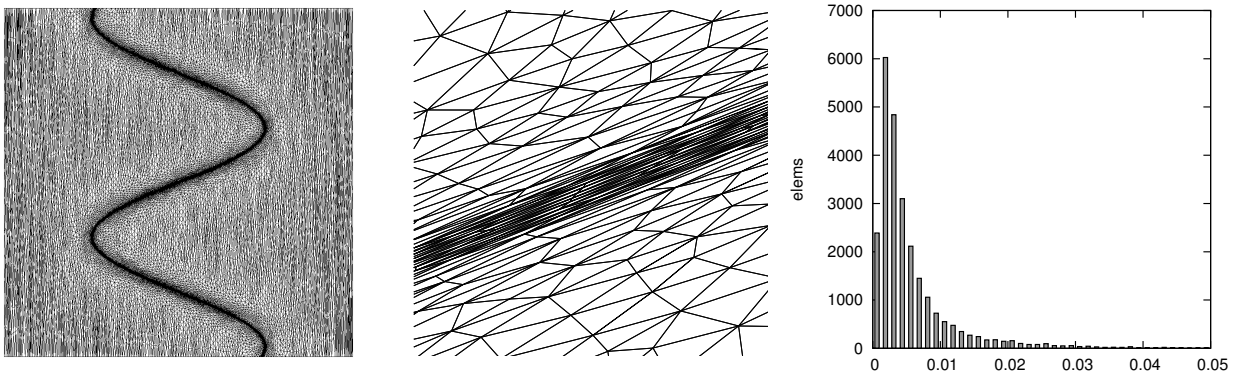
Proof. For each core i and at a given iteration, we have:

- step 1: let $n_{1,i}$ be the number of mesh elements assigned to the core i. Hence $r_i = n_{1,i}$.
  The quality of each element is evaluated, at most 3 values are locally stored in L, so $c_i = \Omega(3n_{1,i})$. A reduction is perfomed on n_L so $\kappa = 2$. At most $n_L \leq \sum_{i=1}^{p} n_{1,i}$ values are written into $\mathbb{L}$ by all cores, so $w_i = \Omega(n_{1,i})$.
- step 2: it refers to graph coloring (see Proposition 2). Let $n_{2,i}$ be the number of vertices of V assigned to the core i. Thus we have $r_i = \Omega(\bar{\Delta}\Bbbk n_{2,i})$, $w_i = \Omega(2\Bbbk n_{2,i})$, $c_i = \Bbbk n_{2,i}$ and $\kappa = 2\Bbbk$, with $\Bbbk$ the number of iterations.
- step 3: let $n_3 = |I|$, and $n_{3,i}$ the number of darts assigned to the core i. The 4 darts $(e_k)_{k=1,4}$ surrounding any e are retrieved locally , so $r_i = 4n_{3,i}$. The flip is done by overwriting references stored in each e_k, and the quality of the 2 resulting elements are calculated. Thus $w_i = 4n_{3,i}$ and $c_i = 2n_{3,i}$. No reduction is done, so $\kappa = 0$.

Thus we have: $t = \sum_{s=1}^{3}[\max_{i=1,p}[\Omega(\Bbbk n_{s,i})] + g\cdot\max_{i=1,p}[\Omega((\Delta+\Bbbk)n_{s,i})] + 2\Bbbk]$ for any scheduling policy.
For a static task partitioning, we have: $n_{s,i} = \Omega(\frac{n_h}{p})$, thus $t = \Omega(\sum_{i=1}^{3}(1+g)(\Bbbk+(\Delta+\Bbbk))\frac{n_h}{p}) = \Omega((1+g)(2\Bbbk+\Delta)\frac{n_h}{p})$, with n_h the number of half-edges. Tasks are equi-distributed and unit task cost is in $\Omega((1+g)(2\Bbbk+\Delta)$.
As $\Bbbk$ is fixed and $\Delta = 4$, thus task cost distribution is uniform. Therefore the load imbalance of a step s is $\iota_s = 0$ □

6. Numerical results

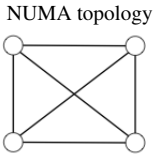
Accuracy. The devised scheme was implemented in C++ using OpenMP 4. An example of mesh adapted to a multi-scale solution field $(u_{\mathsf{p}})_{\mathsf{p}\in\Omega}$ is given in Fig. 7, with $u = \underbrace{0.1\sin(50x)}_{\text{flow}} + \underbrace{\operatorname{atan}(0.1/(\sin(5y) - 2x))}_{\text{shock}}$ on $\Omega = [-1, 1]^2$.
Input mesh: $n = 13\,102$ nodes and $m = 25\,760$ elements, no metric gradation, target number of nodes $N = n$.
Run parameters: 4 threads, $n_a = 5$ adaptations, $n_r = 10$ stages per adapt and $n_s \leq 15$ iterations per stage.
The error is defined as the gap between mesh and the cartesian surface defined by the metric field [24].
It is given by $\varepsilon_{\mathsf{K}} = \max(\varepsilon_{\mathsf{G}_\mathsf{K}}, \varepsilon_{e_\mathsf{K}})$ with $\varepsilon_{\mathsf{G}_\mathsf{K}} = |u_{\mathsf{G}} - \frac{1}{3}\sum_{i=1}^{3} u_{\mathsf{p}_i}|$ and $\varepsilon_{e_\mathsf{K}} = \max_{i=1,3} |u_{e_i} - \frac{1}{2}\sum_{j\neq i} u_{e_j}|$.
All scales of the solution field are correclty captured with a min quality $q_{\mathsf{min}} = 0.512$ and a mean quality $\tilde{q} \approx 0.9$.



(a) adapted mesh (b) zoom in the shock front (c) error distribution

Fig. 7: Mesh adapted to a solution field (u_{p}) defined by $u_{p:(x,y)} = 0.1\sin(50x) + \operatorname{atan}(0.1/(\sin(5y) - 2x))$ on $\Omega = [-1, 1]^2$ with 4 threads.
All scales of the solution field are correclty captured with a min quality $q_{\mathsf{min}} = 0.512$ and a mean quality $\tilde{q} \approx 0.9$.

Strong scaling. Runs were made on a Intel Haswell-EP (32 cores, 4 NUMA nodes, 4GB/core, see Fig. 8).
The code was compiled with Intel compiler suite 15.0.2 with -O3 -march=native flags enabled.
Input data: $n = 504\,100$ nodes and $m = 1\,005\,362$ elements. No index reordering throughout iterations.
Tuning parameters: static round-robin thread binding (1/core), guided scheduling.
Run parameters: $n_a = 3$ adaptations, $n_r = 10$ stages per adaptation, $n_s \leq 15$ iterations per stage.
The shock in Fig. 7 was considered for the adaptation with a target number of nodes $N = n$.



Intel Haswell E5-2698 v3

- 2 sockets, 16 cores / socket, 2.3Ghz
  2 NUMA nodes / socket, 4GB / core
- 3 cache levels, LRU policy
  L1: 32KB / core x 2
  L2: 256KB / core
  L3: 20MB / NUMA node

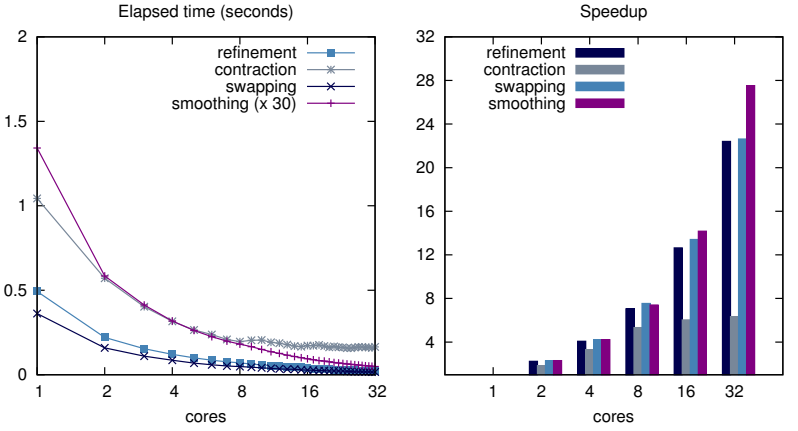


Fig. 8: Mean elapsed time $t = (1/(n_a n_r n_s))\sum_{i=1}^{n_a}\sum_{j=1}^{n_r}\sum_{k=1}^{n_s} t_{i,j,k}$ with $n_a = 3, n_r = 10, n_s \leq 15$, and speedup per iteration.
Refinement and swapping scale well, even on 4 NUMA nodes. Data involved here are mainly local and no substantial load imbalance is expected.
Smoothing scales even better, since its more compute-intensive on one hand, and since primal graph remains unchanged on the other hand.
Contraction suffers from load imbalance due to irregular stencil size distribution, and cache misses penalties.

The mean elapsed time per iteration $t = (1/(n_a n_r n_s)) \sum_{i=1}^{n_a} \sum_{j=1}^{n_r} \sum_{k=1}^{n_s} t_{i,j,k}$ of each remeshing stage is given on Fig. 8. Overall scalability is good with an efficiency of 70% for 3 kernels out of 4 on 32 cores. Refinement and swapping scale well, even on 4 NUMA nodes. Indeed, data involved in these stages are mainly local and no substantial load imbalance is expected, as stated in Proposition 3 (see Table 3). Smoothing stages scale even better. As it is more compute-intensive, data accesses penalties are significantly mitigated. Moreover, it benefits from the reuse of cached cell data, since mesh topology – and the underlying data layout – remains unchanged.
Despite efforts to improve locality, contraction stages suffer from load imbalance due to irregular stencil size distribution (induced by anisotropy), and high last level cache (LLC) misses penalties (see Fig. 10).
The time spent on each step for the refinement, contraction and swapping stage on 32 cores is given in Fig. 9.

stages	bulk-synchronous steps			
	1	2	3	4
refinement	filter	steiner	process	repair
contraction	stencil	graph	process	repair
swapping	filter	graph	process	-

stages	time (ms)			
	1	2	3	4
refinement	26.4	4.6	4.2	7.8
contraction	130.1	3.7	4.2	30.9
swapping	4.1	4.73	16.0	-

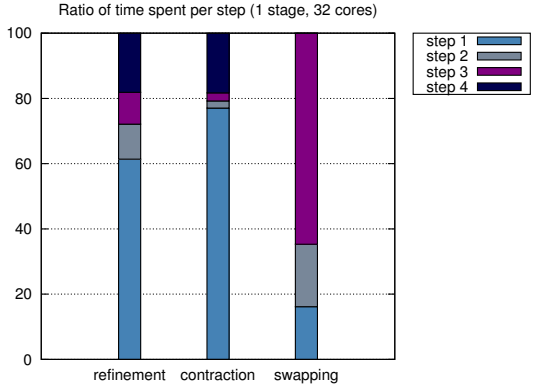


Fig. 9: Ratio of time spent on each bulk-synchronous step for a refinement, contraction and swapping stage on 32 cores.
The overhead spent on task graphs construction and coloring is small compared to other steps, especially for the contraction stage.
In the latter, most of the elapsed time is spent on stencil analysis: the collapse operation is simulated for each target node v, and the validity of the resulting patch is verified at each time (in terms of areas and edge length). Its cost depends on the size and geometrical configuration of the stencil.

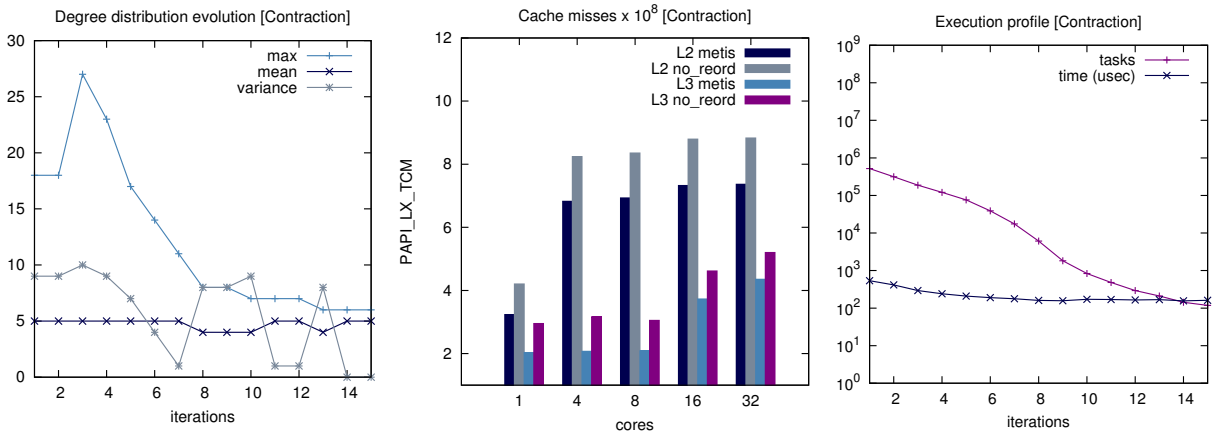


Fig. 10: Degree distribution evolution, cache misses with/without index reordering (using Metis [25] and PAPI counters [26]), and elapsed time profile for one contraction stage on 32 cores. The degree distribution of the primal graph evolves in an irregular way throughout iterations, resulting in load imbalance. Data layout have been significantly evolved throughout iterations, resulting in a high cache misses count.

Further investigation has been done for contraction (see Fig. 10). As pointed in Proposition 3, load imbalance is mainly impacted by degree distribution of mesh primal graph. The deviation on degree distribution of the primal graph shows that stencil sizes evolve in an irregular way throughout iterations. As shown in Fig. 9, most of the elapsed time is spent on stencil analysis step, which is the most irregular part of this stage. Indeed, the collapse operation is simulated for each potential target node v, and the validity of the resulting patch is verified at each time (in terms of element area and edge length). Depending on the size and geometrical configuration of the stencil, this step may involve an important load imbalance. An index reordering using a nested dissection heuristic [25] has been performed

in order to show the impact of data layout on cache performance. It confirms that data layout has been significantly changed throughout iterations, resulting in a high cache misses count.

7. Conclusion and perspectives

A scalable fine-grained lock-free scheme for anisotropic mesh adaptation on NUMA architectures was provided. A mean efficiency of 70% was achieved on 32 cores for 3 kernels out of 4, but further efforts have to be done for the contraction. Two tracks may be considered for this purpose: (1) a locality-aware work-stealing scheme to ease load imbalance and (2) the integration of a fine-grained index reordering stage in the remeshing loop to ease cache penalties (the related overhead is actually the main obstacle).
Further measures (unit task costs – not only for remeshing ones – effective memory bandwidth/latency) have to be done to compare the predicted execution time with the real elapsed time, and to assess the accuracy of the cost model. An extension to a multi-grained scheme (shared/distributed memory) is expected, with the constraint that the bulk-synchronous structure of the algorithm should be kept. A hierarchical bridging model [21] may be used in this case. It will allow to explicitely characterize the latency at each level of the memory hierarchy (caches, local/remote DRAM) for finer prediction. Finally, an extension to the 3D case may be considered.

Acknowledgement

A special thanks to Nicolas Le-Goff for his assistance through all steps of this work.

References

[1] G. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, AFIPS (1967) 483–485.
[2] T. Gautier, et al., Regular Versus Irregular Problems and Algorithms, IRREGULAR (1995) 1–25.
[3] J.-F. Remacle, et al., A Two-Level Multithreaded Delaunay Kernel, IMR 24 (2015) 6–17.
[4] N. Chrisochoides, A Survey of Parallel Mesh Generation Methods, Brown University, Providence (2005).
[5] A. Loseille, et al., Parallel Generation of Large-Size Adapted Meshes, IMR 24 (2015) 57–69.
[6] C. Antonopoulos, et al., A Multigrain Delaunay Mesh Generation Method for Multicore SMT-based Architectures, JPDC (2009) 589–600.
[7] L. Freitag, et al., The Scalability Of Mesh Improvement Algorithms, IMA Volumes in Maths and its Applications (1998) 185–212.
[8] G. Rokos, et al., Thread Parallelism for Highly Irregular Computation in Anisotropic Mesh Adaptation, EASC (2015) 103–108.
[9] H. Borouchaki, et al., Mesh Gradation Control, Numerical Methods in Engineering (1998) 1143–1165.
[10] F. Alauzet, et al., Multi-Dimensional Continuous Metric for Mesh Adaptation, IMR 15 (2006) 191–214.
[11] M.-G. Vallet, et al., Numerical Comparison of Some Hessian Recovery Techniques, Numerical Methods in Engineering 72 (2007) 987–1007.
[12] Y. Vassilevski, K. Lipnikov, Adaptive Algorithm For Generation of Quasi-optimal Meshes, Computational mathematics (1999) 1532–1551.
[13] J. Lee, et al., When Prefetching Works, When It Doesn't, and Why, ACM Trans. on Architecture and Code Optimization (2012).
[14] F. Kramer, H. Kramer, A Survey on the Distance-Colouring of Graphs, Discrete Mathematics 308 (2008) 422–426.
[15] J. R. Allwright, et al., A Comparison of Parallel Graph Coloring Algorithms, 1995.
[16] U. Çatalyurek, et al., Graph Colouring Algorithms for Multicore and Massively Multithreaded Architectures, JPC (2012) 576–594.
[17] D. Chakrabarti, et al., R-MAT: A Recursive Model for Graph Mining, SIAM Data Mining (2004) 442–446.
[18] A. Gebremedhin, Practical Parallel Algorithms for Graph Coloring Problems in Numerical Optimization, Ph.D. thesis, Univ. Bergen, 2003.
[19] G. Damiand, Contributions aux Cartes Combinatoires et Cartes Généralisées, HDR, INSA de Lyon, 2010.
[20] L. Valiant, A Bridging Model for Parallel Computation, ACM Comm. 33 (1990) 103–111.
[21] L. Valiant, A Bridging Model for Multicore Computing, JCSS 77 (2011) 154 – 166.
[22] V. Ramachandran, et al., Can a Shared-Memory Model Serve as a Bridging Model for Parallel Comp.?, Theory of Comp. Sys. (1999) 327–359.
[23] B. Grayson, et al., Experimental Evaluation of QSM, a Simple Shared-Memory Model, IPPS/SPDP (1999) 130–136.
[24] F. Alauzet, Adaptation de Maillage Anisotrope en 3D. Application aux Simulations Instationnaires en Méca. des Fluides, Ph.D. thesis, 2003.
[25] D. LaSalle, G. Karypis, Efficient Nested Dissection for Multicore Architectures, EuroPAR (2015) 467–478.
[26] P. J.Mucci, et al., PAPI: A Portable Interface to Hardware Performance Counters, Dep. of Defense HPCMP Users Group Conf. (1999) 7–10.