Scalable Fine-grained Metric-based Remeshing Algorithm for Manycore/NUMA Architectures

Hoby Rakotoarivelo[1,2]✉, Franck Ledoux[1], Franck Pommereau[2], and Nicolas Le-Goff[1]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
{franck.ledoux,nicolas.le-goff}@cea.fr
[2] IBISC, Université d'Évry Val d'Essonne, France
{hoby.rakotoarivelo,franck.pommereau}@ibisc.fr

Abstract. In this paper, we present a fine-grained multi-stage metric-based triangular remeshing algorithm on manycore and NUMA architectures. It is motivated by the dynamically evolving data dependencies and workload of such irregular algorithms, often resulting in poor performance and data locality at high number of cores. In this context, we devise a multi-stage algorithm in which a task graph is built for each kernel. Parallelism is then extracted through fine-grained independent set, maximal cardinality matching and graph coloring heuristics. In addition to index ranges precalculation, a dual-step atomic-based synchronization scheme is used for nodal data updates. Despite its intractable latency-boundness, a good overall scalability is achieved on a NUMA dual-socket Intel Haswell and a dual-memory Intel KNL computing nodes (64 cores). The relevance of our synchronization scheme is highlighted through a comparison with the state-of-the-art.

Keywords: Irregular parallelism, Manycore, Anisotropic remeshing.

1 Introduction

In computational fluid dynamics, large-scale direct numerical simulations require a high discretization (mesh) resolution to achieve a good accuracy. Moreover, the computational domain needs to be periodically re-discretized to avoid degenerated or mixed cells in case of lagrangian-based or multi-materials simulations[4]. In this context, triangular mesh adaptation aims at reducing the computational effort of these simulations while preserving the required accuracy. However, its parallelization remains challenging due to dynamically evolving data dependencies and workload, resulting in a poor locality and efficiency at high number of cores. On the other hand, manycore architectures have been emerged in HPC landscape, with an increasing number of cores but a decreasing memory and frequency per core, and an asymmetric memory latency in case of NUMA multi-socket machines. To take advantage of these architectures, the challenge is to expose a high concurrency and data locality for such an irregular algorithm.

RELATED WORKS Most of existing parallel remeshing schemes are coarse-grained, and not suitable to manycore machines. They rely on domain partitioning and dynamic cell migration for load balancing. They focus on reducing the unavoidable synchronization for domain interface consistency, and on finding reliable heuristics for cell migration [7]. Fine-grained schemes have emerged but most of them rely on a speculative execution model [2, 5]. In 2015, Rokos *et al.* devised a clean lock-free scheme in [11, 12], based on an initial idea of Freitag *et al.* [6]. Task conflicts are expressed by a graph, and non-conflictual tasks are then explicitely extracted. To avoid data races, mesh data updates are stacked locally and committed later. Their solution scaled well on a dual-socket SandyBridge machine, but worse on a quad-socket Opteron one due to NUMA effects. Indeed, data placement is not taken into account on tasklists reduction. Furthermore, their deferred updates scheme involves a lot of data moves, increasing NUMA effects while reducing the arithmetic intensity. In [10], we extended their work by using kernel-specific graphs to increase parallelism, and a combinatorial map [3] data structure to avoid synchronization for mesh data updates. We attempted a theoretical characterization of performance metrics, based on machine parameters (e.g bandwidth). Our solution scaled well on a dual-socket Haswell machine, but the contraction kernel suffers from memory indirections on stencil data retrieval.

CONTRIBUTIONS This paper is an extension of our preliminary work in [10]. It differs from [10–12] in many points:

1. We use a dual-step atomic-based synchronization scheme for topological updates, with a node-centered data structure. We show that it is a good tradeoff between data locality and synchronization cost (overhead, data moves). This way, we improve the efficiency of the contraction kernel which was the main drawback in [10]. We also show that taking into account the graph number of connected components would increase the parallelism for this kernel.
2. We use a fine-grained maximal graph matching heuristic for task extraction in the swapping kernel. We are the first to apply such a scheme in parallel meshing and we show that it is efficient in practice.
3. Evaluations are made on both a NUMA dual-socket and a dual-memory machines. Our results show that the latency-boundness of such an algorithm is intractable due to its high irregularity, but may be eased by the use of hyperthreading. Such an evaluation was not yet done on Intel Xeon-Phi KNL in parallel meshing context.

2 Problem overview

The purpose is to rebuild a discretization (mesh) of a domain Ω, such that the interpolation error of a given solution field u is bounded and equi-distributed on Ω. It is done by an iterative procedure, and involves a numerical solver and a metric-based remesher. It ends when a given error treshold is achieved (Algorithm 1). In our context, a node refers to a mesh point and a cell refers to a mesh triangle.

REMESHING To control the interpolation error of u, cells size and density must fit the variation of the physical solution field over the domain. Basically, it may be achieved by three ways:

- *variational:* node sampling is obtained by minimizing an energy function, and resulting nodes are then triangulated using a Delaunay kernel.
- *hyperspace embedding:* nodal coordinates, solution field and related gradient are embedded in $\mathbb{R}^6$. The domain is then remeshed in this hyperspace, using local or global kernels.
- *metric-based:* a tensor field is associated to each node v_i, and encodes cell size and stretching (anisotropy) prescription in the vicinity of v_i. An uniform mesh is then built in the riemannian metric space induced by the tensor field.

We opt for a metric-based scheme since it is local and preserve well anisotropy compared to the two others. A standard sequence of operations is used for that purpose. First, we compute a nodewise tensor field from nodal discrete second derivatives. A gradation is then performed to smooth out sudden changes in size requirements. Afterwards, we apply the geometric and topological operations on mesh, using 4 local kernels:

- the *refinement* which aims at splitting long edges by recursive cell dissection.
- the *contraction* which aims at collapsing short edges by vertex merging.
- the *swapping* which improves cell pair qualities by edge flips.
- the *smoothing* which improves stencil qualities by relocating nodes using an anisotropic laplacian operator.

Algorithm 1 Adaptive loop

```
input: mesh, error and quality tresholds.
output: optimal couple mesh-solution.
  repeat
    solve the solution field (u_p) on mesh.
    derive a tensor field from (u_p).
    apply gradation on tensor field.
    while min. quality not optimal do
      refinement
      contraction
      swapping
      smoothing
    end while
  until error threshold is reached
  return couple mesh-solution
```

Algorithm 2 Kernel parallel stages

```
repeat
  1. filter active nodes/cells.
  2. build a task graph G = (V, E)
  3. extract non-conflictual tasks
  4. apply operations
  5. repair topology
until no marked cells
```

PARALLELIZATION Remeshing is a data-driven algorithm. Tasks (gradation, refinement, contraction, swapping, smoothing) are related to a dynamically evolving subset of nodes or cells. In fact, processing a task may generate some others, and need to be propagated. In our case, the required number of rounds is data-dependent. Finally, tasks within a same round may be conflictual [9].

Here, data dependencies are related to mesh topology, and evolves accordingly:

- gradation, contraction and smoothing involve the vicinity of each active node;
- refinement involves a subset of the vicinity of each active cell;
- swapping involves the vicinity of each active cell pair.

In fact, two issues must be addressed: topological inconsistency and data races. Indeed, conflictual tasks may invalid the mesh (crosses, holes or boundary loss) whereas nodal or incidence data may be corrupted if updates are not synchronized. The former is solved by an explicit parallelism extraction (Section 3), whereas the later is solved by an explicit synchronization scheme (Section 4).
Kernels are parallelized independently using a fork-join model. Each of them iteratively performs 5 stages (Algorithm 2). Here, any data updated in a given stage cannot be used within the same stage.

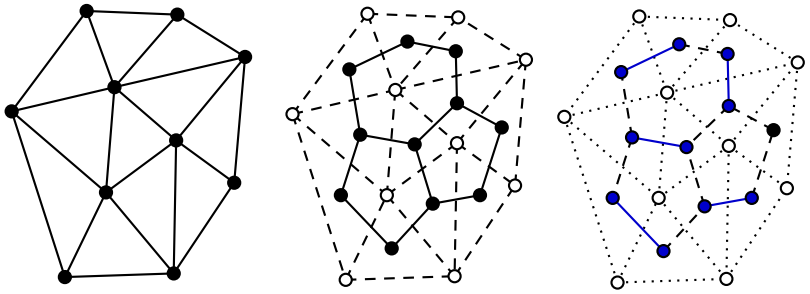
3 Extracting fine-grained parallelism

For each kernel, we extract a task graph $G = (V, E)$. Their descriptions are given in Figure 1. However, no graph is required for refinement since cells may be processed asynchronously. V is a set of active tasks, and E represents task conflicts. Parallelism is then extracted through fine-grained graph heuristics (Table 1).

Table 1: Task graphs per kernel and related heuristics

kernel	graph extracted from	heuristic
gradation	mesh primal graph	coloring
refinement	none	–
contraction	mesh primal graph	indep. set
swapping	mesh dual graph	matching
smoothing	mesh primal graph	coloring

CONTRACTION For each topological update, mesh conformity must be preserved such that holes and edge crosses are avoided. However, collapsing two neighboring nodes may result in a hole, so they cannot be processed concurrently. Thus, the idea is to extract independent nodes such that they can be processed in a safe way. For it, we derived a heuristic from a graph coloring scheme in [1]. Here, the number of connected components σ_G of G increase through iterations. In our case, we always pick the lowest available color according to neighbors values, then the ratio of independent tasks increases according to σ_G. We resolve conflicts only for the first color to accelerate the procedure. Also, tie breaks are based on vertex degree (Algorithm 3). A comparison with a monte-Carlo based heuristic [8] shows that taking the variation of σ is relevant in our context. Indeed, the ratio of independent nodes on $|V|$ is greater in this case (Figure 2).



(a) mesh = primal graph (b) mesh dual graph (c) cell matching

Fig. 1: Graph descriptions and cell pair matching in swapping kernel

SWAPPING Flipping more than one edge per cell may result in an edge cross. For each cell K_i, the unique edge e to be flipped, and thus the neighboring cell K_j sharing e, must be identified. Therefore, we aim at extracting a subset of cell pairs to be flipped. For it, the idea is to extract a maximal cardinality matching from the dual graph (Figure 1). To do that, we adapt the Karp-Sipser's heuristic. It is based on vertex-disjoint augmenting path retrievals using depth first searches in G (Algorithm 4). Here, it is irrelevant to maintain different tasklist according to cell degrees, since we know that they are whether 2 or 3. The ratio of matched cells shows that this greedy scheme is convenient for our purposes (Figure 2).

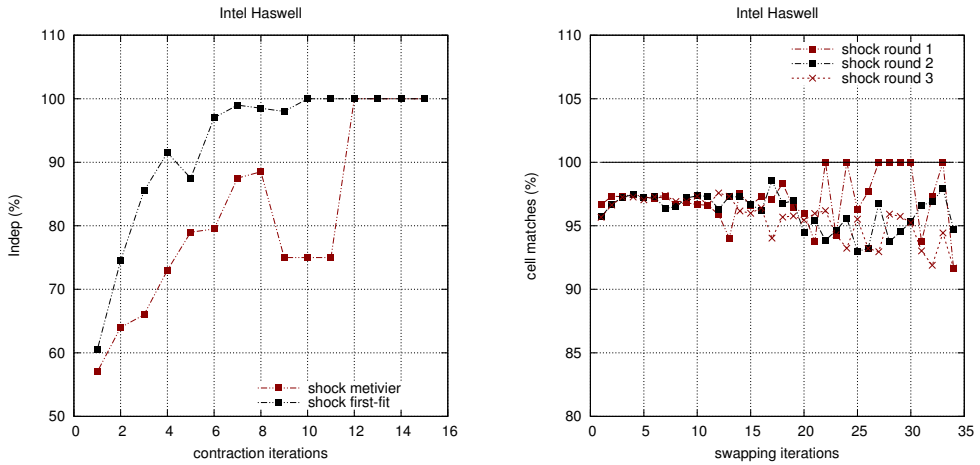
Algorithm 3 Nodes extraction

```
U ← V
repeat
    ∀v ∈ U, select smallest available
    color according to N_v
    (do not care about data races)
    for vertex v ∈ U in parallel
        if col[v] = 1, ∃w ∈ N_v, col[w] = 1
            if deg[v] ≥ deg[w] then
                add v to R
    U ← R, R ← ∅
until U = ∅
return I ← {v | col[v] = 1}
```

Algorithm 4 Cell matching

```
M ← ∅, visited[] ← {0}
for marked cell K in parallel
    S ← {K}                          ▷ local stack
    repeat
        u ← S[0]
        if compare_swap(visited[u])=0 then
            for cell v ∈ N_K do
                if compare_swap(visited[v])=0
                    add (u, v) to M
                    for w ∈ N_v do
                        if fetch_sub(deg[w],1)=2
                            push w in S
    until S ← ∅
return M
```

GRADATION *and* SMOOTHING In these kernels, the computed value of a given node v_i is interpolated from its neighbors $\mathcal{N}[v_i]$. However, processing v_i and any



(a) Ratio comparison of our heuristic (First-fit) and a Monte-Carlo based scheme (Metivier) on independent nodes extraction.

(b) Ratio of matched cells throughout swapping rounds while |V| is decreasing linearly

Fig. 2: Heuristics performance on tasks extraction for contraction and swapping kernels

$v_j \in \mathcal{N}[v_i]$ may result in data races. Thus, we aim at extracting a nodal partition such that no two neighboring nodes will be scheduled concurrently. For that, we use a fine-grained graph coloring in [1]. In our case, kernels convergence rate decreases linearly on the number of colors. Thanks to the planarity of the graph, the practical number of colors remains low (between 5 and 7).

4 Synchronizing for topological updates

In our case, mesh topology is explicitely stored by maintaining incidence lists. Here, extracting non-conflictual tasks does not avoid data races on topological data updates. To resolve it, we define an explicit thread synchronization scheme.

CELL INSERTIONS For the sake of spatial data locality, we store mesh data in shared flat arrays. Since we don't use the same pattern for refinement, then the number of nodes and cells to be inserted cannot be predicted. They can be stacked locally before being globally copied like in [11, 12], but it would result in a high amount of data moves. Instead we infer the number of new cells in order to find the right index range per thread. First, we store the pattern to be applied for each cell in an array `pattern` during the filtering step. Then, each thread t_i performs a reduction on `pattern` within its iteration space $(n/p) \cdot [i, i+1]$, with n the number of tasks and p the number of threads. The result is then stored in an array `offset[`i`]`. Finally a prefix-sum is done on `offset` to retrieve the right index ranges per thread for cell insertions.

Table 2: Deferred updates mechanism in [11, 12]
Thread t_i stores data of node v_k in def_op[i][j] list, with j=hash(k) % p.
Finally, each thread t_i copy all data in $\text{def_op[k][i]}_{\text{k=1,p}}$ in mesh

		updates processed by				
		t_0	t_1	t_2	$\cdots$	t_{n-1}
committed by	t_0	def_op[0][0]	def_op[1][0]	def_op[2][0]	$\cdots$	def_op[n-1][0]
	t_1	def_op[0][1]	def_op[1][1]	def_op[2][1]	$\cdots$	def_op[n-1][1]
	t_2	def_op[0][2]	def_op[1][2]	def_op[2][2]	$\cdots$	def_op[n-1][2]
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	t_{n-1}	def_op[0][n-1]	def_op[1][n-1]	def_op[2][n-1]	$\cdots$	def_op[n-1][n-1]

INCIDENCE DATA UPDATES We use a node-centered data structure: each node stores the index of cells connected to it. Here, updates may be stacked locally using the deferred mechanism in [11, 12] (Table 2). Due the huge amount of data moves, it would increase significantly the overhead of this step (Figure 8). We use a dual-step synchronization scheme instead. First, cells indices are added asynchronously in nodal incidence lists (Algorithm 5). For that, threads increment atomically a nodal offset array deg. Each node is then atomically marked as to be fixed, since its incidence list may contain obsolete references. Finally, incidence lists of each marked node are fixed in a separate step (Algorithm 6).

Algorithm 5 step 1: asynch. adds

```
input: data, n
  atomic_compare_swap(fix[i], 1)
  k ← atomic_fetch_add(deg[i], n)
  if n + k exceeds incid[i] capacity then
  |  #pragma omp critical
  |  /* double check pattern */
  |  if not yet reallocated then
  |  |  realloc incid[i] to twice its capacity
  end if
  copy data to incid[i][k]
```

Algorithm 6 step 2: repair sweep

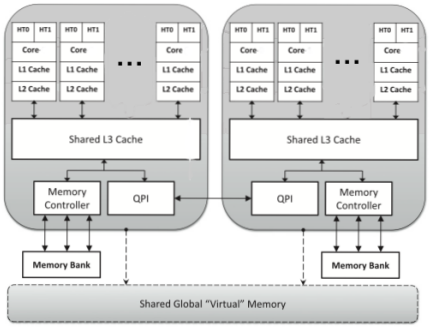
```
R ← ∅                                  ▷ local to thread
for node v_i in mesh in parallel
  |  if fix[i] then
  |  |  for cell K in incid[i] do
  |  |  |  if v_i ∈ K then
  |  |  |  |  add K in R
  |  |
  |  |  incid[i] ← ∅, deg[i] ← |R|
  |  |  sort R and swap with incid[i].
  |  end if
```

5 Evaluation

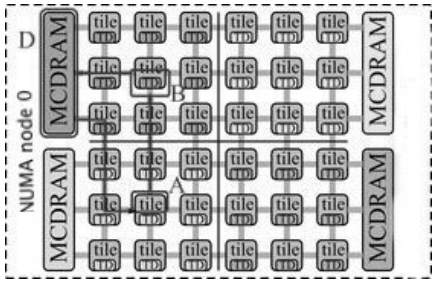
Our algorithm is implemented in C++ using OpenMP4 and C++11 capabilities.

BENCHMARK PARAMETERS Our code is compiled with the Intel compiler suite (version 15) with -O3 flag enabled. Thread-core affinity is done by setting the environment variable KMP_AFFINITY to scatter on normal mode and compact on

hyperthreading. Benchmarks are run on a NUMA dual-socket Intel Xeon Haswell E5-2698 v3 machine (2x16 cores at 2.3Ghz, 3 cache levels), and an Intel Xeon-Phi KNL machine (72 cores at 1.5Ghz, 2 cache levels, 4 HT/core). KNL has two memory: an on-chip MCDRAM at 320 GB/s and a DDR4 at 60 GB/s. We use the quadrant clustering mode to ease cache misses worst case penalties (Figure 3).



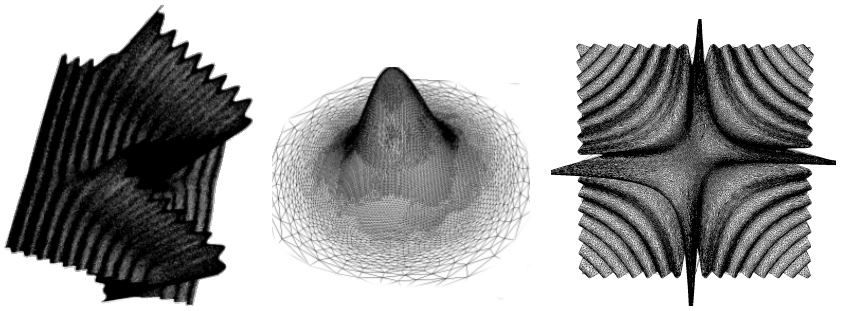
(a) Cache/memory hierarchy in Haswell. L2/L3 cache latencies ≈ 4.7 ns and 6.4 ns. Local and remote memory ≈ 18 ns, 40 ns.



(b) KNL quadrant clustering mode. Each tile consists of a dual-core and a shared L2 cache. Physical addresses are mapped to tag directories such that memory requests do not need to go across quadrants.

Fig. 3: Cache and memory organization in Intel Haswell and KNL computing nodes.

We use 3 solution fields with different anisotropy levels for our tests (Figure 4). For each testcase, an input grid of 1 005 362 cells and 504 100 nodes is used. It is initially renumbered by a Hilbert space-filling curve scheme, but no reordering is done during the execution. For each run, a single adaptation is performed with 3 rounds. Mesh density factor is set to 0.9, and no metric gradation is performed.



(a) shock (b) gauss (c) waves

Fig. 4: Solution fields used in our benchmarks

STRONG SCALING The mean makespan and scaling efficiency $E_\mathsf{p} = t_1/(\mathsf{p} \cdot t_\mathsf{p})$ are given in Figure 5, with t_n the makespan on n threads. Hyperthreading is systematically used on KNL (4 per core) to hide memory access latency (30 and 28 ns for MCDRAM and DDR4 respectively). We use both MCDRAM and DDR4 by binding memory through `numactl`. All testcases behaves similarly and a good scaling is achieved on both architectures. Surprisingly, there was no significant improvement in the use of the high bandwidth MCDRAM. Indeed, the algorithm is not bandwidth-sensitive. The efficiency falls to 30% on KNL on 256 threads due to high contentions, but scales better than on Haswell on lower number of cores. Makespan is still improved when using hyperthreading on Haswell, and NUMA effects are significantly eased, thanks to a locality-aware data updates scheme.

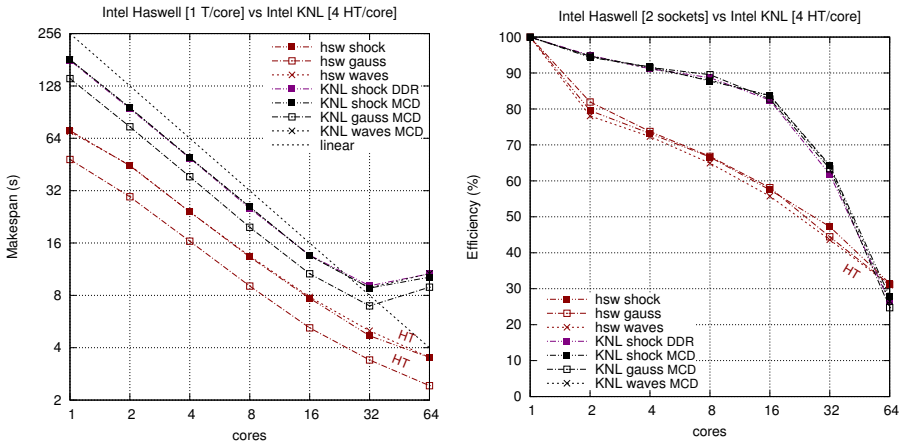


Fig. 5: Mean makespan and scaling efficiency on 3 rounds.

OVERHEADS PER KERNEL For each kernel, the time spent distribution per step is given in Figure 6. Overheads related to parallelism extraction and synchronization are depicted in red. These steps do not exceed 15% of the makespan for all kernels. Furthermore, they are negligible in case of contraction and smoothing. Also, these ratios remain constant despite the number of threads, and scale at the same rate as other steps. For the refinement, operations are structured such that no parallelism extraction is required. Moreover, the filtering step does not require a full consistent mesh topology for adjacency requests, in the sense that stored incidence lists may contain obsolete references, but each new cell $\mathsf{K} : (v_0, v_1, v_2)$ must be referenced in incidence lists of $(v_i) \in \mathsf{K}$. For this kernel, the repair sweep involved in the synchronization scheme is performed once at the very end of the procedure. For the contraction, the vicinity $\mathcal{N}[v_i]$ of each node v_i is required by the filtering step in order to find the right $v_j \in \mathcal{N}[v_i]$ where v_i should collapse to (even in sequential). Hence, the primal graph is recovered

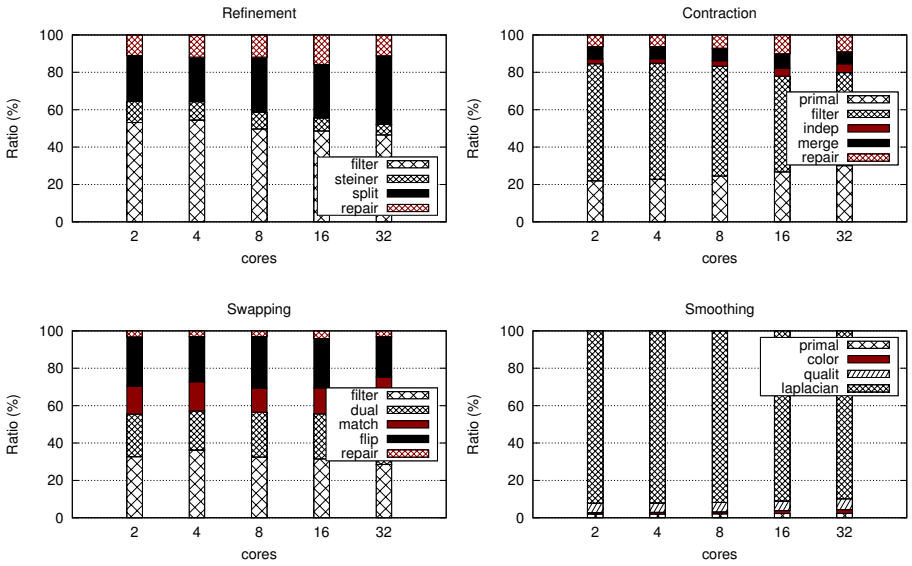


Fig. 6: Time ratio distribution per step for each kernel on Haswell.

at the beginning of each round. This step mainly consist of data accesses but represents roughly 22% of the overall makespan. It involved a high amount of cache misses in [10] due to memory indirections when requesting the combinatorial map data structure. In our case, stencil retrieval involves only one level of indirection (instead of two), leading to a better scalability (Algorithm 7 and 8). For swapping, the main overhead is related to graph matching stage with a mean ratio of 15%. Its convergence is linear to the search depth δ_G on augmenting paths retrievals. This step is highly irregular and is asymptotically in $\mathcal{O}(\log n)$ with n the number of vertices of the dual graph. In practice, $\delta_G \approx 4$ with static scheduling, and nearly 12 rounds is required for step convergence.
For smoothing, the primal graph is recovered at the beginning of the procedure, but no synchronization sweep is required since mesh topology remains unchanged. For this kernel, the unique overhead is related to the graph coloring step. In practice, a low number of rounds is required for convergence (roughly 3).

Algorithm 7 stencil retrieval of v_i

for each $\mathsf{K} : (\mathrm{p}_0, \mathrm{p}_1, \mathrm{p}_2) \in \mathsf{incid}[v_i]$ **do**
  if $\mathrm{p}_i = v_i$ **then**
    $(j, k) : (i{+}1 \mod 3, i{+}2 \mod 3)$
    add $\mathrm{p}_j, \mathrm{p}_k$ in $\mathcal{N}[v_i]$
sort $\mathcal{N}[v_i]$ and remove duplicates

Algorithm 8 stencil retrieval in [10]

$\mathsf{init} \leftarrow v_i.\mathsf{edge}$; $\mathsf{cur} \leftarrow \mathsf{init}$
repeat
  add $\mathsf{cur}.\mathsf{v2}$ to $\mathcal{N}[v_i]$
  $\mathsf{inv} \leftarrow \mathsf{opp}[\mathsf{cur}]$, $\mathsf{cur} \leftarrow \mathsf{next}[\mathsf{inv}]$
until $\mathsf{cur} = \mathsf{init}$

PERFORMANCE PER KERNEL Task and floating-point operations (FLOP) rates per kernel on Haswell are given in Figure 7. Refinement and swapping have higher task rates since they are both much local and less compute-intensive than the two others. Refinement involves the vicinity $\mathcal{N}[\mathsf{K}]$ of each cell K, because when an edge is split then the two surrounding cells are dissected. However, the dissection step is purely local because the index of the node to be inserted is already resolved in the steiner point computation step. Therefore, cells may be dissected individually. Swapping steps involve the shell of each edge[3] to be flipped which size is constant, whereas contraction and smoothing steps involve the stencil $\mathcal{N}[v_i]$ of each node v_i, whose size is variable and related to anisotropy.

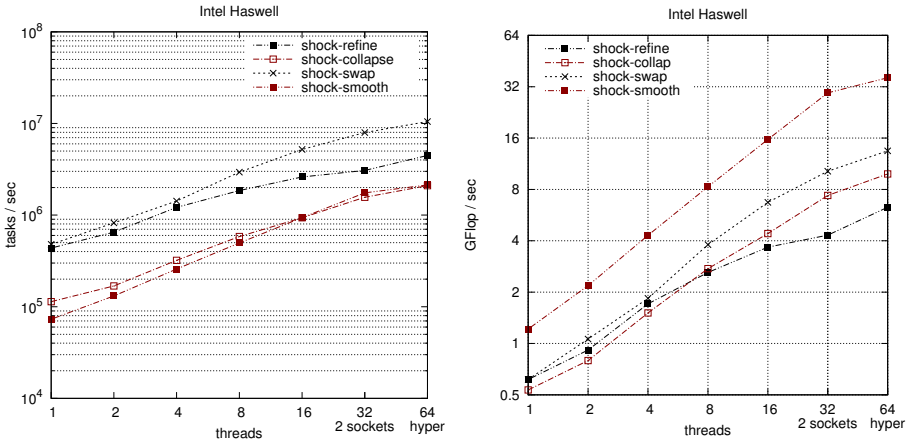


Fig. 7: Task rate and floating-point operations per second of each kernel on Haswell

All kernels scale well in terms of FLOP rate. Most of floating-point operations occur during the filtering step, except for smoothing. This step involves geodesic distance calculation for refinement and contraction, and cell quality computation sweep for swapping and smoothing. In our case, the arithmetic intensity (the ratio of FLOP on the amount of data accesses) remain roughly constant with respect to the number of threads. Thus data-movement involved by the synchronization scheme has not a significant impact on FLOP rate, even on higher number of threads. Smoothing scales even better since it has higher arithmetic intensity. In the anisotropic laplacian computation step, coordinates and metric tensor of a given node v_i are interpolated from those of its vicinity $\mathcal{N}[v_i]$, and reajusted iteratively such that v_i remain inside the geometrical convex hull of $\mathcal{N}[v_i]$. Thus, it involves a better reuse of cached data.

[3] The two cells $(\mathsf{K}_1, \mathsf{K}_2)$ sharing this edge, and the stencil $\mathcal{N}[v_k]$ of each $v_k \in \mathsf{K}_i$

SYNCHRONIZATION COST We faithfully implemented the deferred mechanism used in [11, 12] (Table 2) in order to compare it with our dual-step atomic-based synchronization scheme for nodal data updates. To reduce NUMA effects, first-touch policy is applied and no memory reallocation is performed on both cases. Makespan and related overheads are given in Figure 8 on Haswell and KNL. Both schemes scale well, but makespan has doubled in case of the deferred update scheme. In this case, data movement overhead has a significant impact on total execution time of the algorithm. Indeed, deferred mechanism overheads are 5 times the overhead of our synchronization scheme for contraction and swapping kernels.

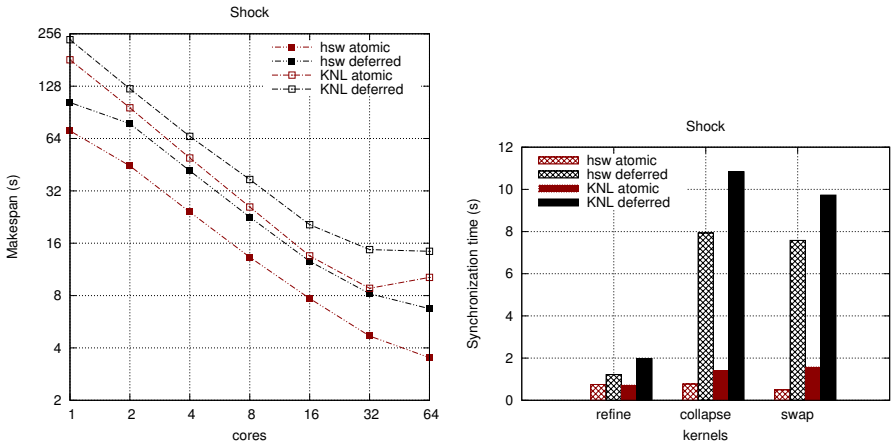


Fig. 8: Cost comparison of atomic-based and deferred synchronization schemes

6 Conclusion

A fine-grained multi-stage algorithm for anisotropic triangular mesh adaptation on manycore and NUMA architectures is proposed. It is based on explicit parallelism extraction using fine-grained graph heuristics, and a dual-step synchronization scheme for topological data updates. It follows a fork-join model and is implemented in C++ using OpenMP4 and auto-vectorization capabilities. Its scalability is evaluated on both a NUMA dual-socket Intel Haswell and dual-memory Intel KNL computing nodes. A good overall scalability is achieved since a mean efficiency of 48% and 65% is reached on Haswell and KNL on 32 cores. Due to higher contentions, a lower efficiency (roughly 30%) is achieved on KNL on 64 cores with 256 threads. Task rate and floating-point operations per second scale in a nearly linear way for all kernels. Overheads related to parallelism extraction as well as synchronization do not exceed 15% of overall makespan.

They remain negligible for contraction and smoothing (5-7%), and scale linearly on other stages makespan. Further efforts have to be done to reduce the latency-sensitiveness of the algorithm, and to take advantage of the high bandwidth on-chip MCDRAM in KNL. Also, a comparison with a task-based version with work-stealing capabilities would be interesting. It would highlight performance sensitiveness if whether data locality is privileged at expense of load imbalance and vice-versa. An extension to a distributed-memory scheme is expected, with a constraint that the bulk-synchronous property of the algorithm should be preserved. In this case, a multi-bulk synchronous parallel bridging model [13] may be used to theoretically characterize its performance, given bandwidth and latency at each level of the memory hierarchy.

References

1. Çatalyurek, U., et al.: Graph colouring algorithms for multicore and massively multithreaded architectures. JPC pp. 576–594 (2012)
2. Chrisochoides, et al.: A multigrain Delaunay mesh generation method for multicore SMT-based architectures. JPDC pp. 589–600 (2009)
3. Damiand, G., Lienhardt, P.: Combinatorial maps: Efficient data structures for computer graphics and image processing. A.K.Peters (2014)
4. Del Pino, S.: Metric-based mesh adaptation for 2D Lagrangian compressible flows. JCP pp. 1793–1821 (2011)
5. Foteinos, et al.: High quality real-time image-to-mesh conversion for finite element simulations. ICS 27 pp. 233–242 (2013)
6. Freitag, et al.: The scalability of mesh improvement algorithms. IMA Maths pp. 185–212 (1998)
7. Loseille, et al.: Parallel generation of large-size adapted meshes. IMR 24 pp. 57–69 (2015)
8. Métivier, Y., et al.: An optimal bit complexity randomized distributed MIS algorithm. JDC pp. 331–340 (2011)
9. Pingali, et al.: Amorphous data-parallelism in irregular algorithms. Tech. Rep. 09-05, University of Texas (2009)
10. Rakotoarivelo, et al.: Fine-grained locality-aware parallel scheme for anisotropic mesh adaptation. IMR 25 pp. 123–135 (2016)
11. Rokos: Scalable multithreaded algorithms for mutable irregular data with application to anisotropic mesh adaptivity. Ph.D. thesis, Imperial College London (2014)
12. Rokos, et al.: Thread parallelism for highly irregular computation in anisotropic mesh adaptation. EASC pp. 103–108 (2015)
13. Valiant, L.: A bridging-model for multicore computing. JCSS pp. 154–166 (2011)