A PARALLEL EDGE ORIENTATION ALGORITHM FOR QUADRILATERAL MESHES *

M. HOMOLYA†‡ AND D. A. HAM‡§

Abstract. One approach to achieving correct finite element assembly is to ensure that the local orientation of facets relative to each cell in the mesh is consistent with the global orientation of that facet. Rognes et al. have shown how to achieve this for any mesh composed of simplex elements, and deal.II contains a serial algorithm to construct a consistent orientation of any quadrilateral mesh of an orientable manifold.

The core contribution of this paper is the extension of this algorithm for distributed memory parallel computers, which facilitates its seamless application as part of a parallel simulation system. Furthermore, our analysis establishes a link between the well-known Union-Find algorithm and the construction of a consistent orientation of a quadrilateral mesh. As a result, existing work on the parallelisation of the Union-Find algorithm can be easily adapted to construct further parallel algorithms for mesh orientations.

Key words. finite element assembly; parallel algorithm; quadrilateral mesh; Firedrake; ARCHER

AMS subject classifications. 68W10; 68W15

1. Introduction. A characteristic of the finite element method for the solution of partial differential equations is that the representation of functions over the domain can be chosen from a wide range of function spaces. This choice is achieved by selecting a particular local function space to be used for each cell in the meshed domain. Information is communicated between adjacent cells either by shared degrees of freedom on the cell boundaries, or by flux integrals over the facets between cells.

For every case except the lowest degree continuous finite elements, these coupling terms require the orientations of the two cells adjacent to each facet to be reconciled with the orientation of that facet. Failure to do so will result in false identification of degrees of freedom falling on the facets, or incorrect accounting for the flux direction through facets.

Consequently, every finite element implementation which supports facet integrals, elements of polynomial degree greater than one, or \( H(\text{div}) \) or \( H(\text{curl}) \) conforming elements must somehow ensure that adjacent cells agree on the orientation of the intervening facet. This can either be achieved by explicitly recording orientation information and exploiting this in the local and/or global assembly operations, or it can be achieved by ensuring, in a sense which we will later make formal, that the local orientation of facets relative to each cell in the mesh is consistent with the global orientation of that facet.

The consistent numbering approach has the advantage that the integral assembly code is simpler than that required by the explicit orientation approach, at the cost of somehow establishing the consistent numbering and with the limitation that a consistent numbering may not exist for all meshes. Consistent numberings are simple to

---

*This work was supported by The Grantham Institute and the National Environmental Research Council [grant numbers NE/K006789/1 and NE/K008951/1]. This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

†The Grantham Institute, Imperial College London, London, SW7 2AZ, UK
‡Department of Computing, Imperial College London, London, SW7 2AZ, UK
§Department of Mathematics, Imperial College London, London, SW7 2AZ, UK

1By facet we mean a mesh entity of codimension 1. In two dimensions these are the cell edges, in three dimensions, the cell faces.
construct for any mesh composed of simplex elements [21] but the same algorithm does not extend to quadrilaterals. Instead, the deal.II finite element library [5] contains an algorithm to construct a consistent orientation of any quadrilateral mesh of an orientable manifold\(^2\). The algorithm employed in deal.II is serial. This is less than ideal on modern supercomputers, for which it would be both more convenient and more efficient to construct the orientation using a distributed parallel algorithm. The core contribution of this paper is therefore to present a distributed parallel extension of the algorithm in deal.II which creates a global consistent numbering. This algorithm is presented in §5, with experimental evaluation in §6. The formal definition of the problem is given in §2 and the proof of the existence of a solution in §3. §4 restates the deal.II serial algorithm in terms of this proof. The parallel part of this algorithm is, in fact, an adaption of the well-known Union-Find algorithm, so in §7 we place this algorithm in the context of that existing work. We conclude the paper in §8.

Facet orientation has been addressed in different ways by various finite element packages. Rognes et al. [21] give a more detailed discussion of the features which make consistent orientations useful for the efficient assembly of finite elements. They also provide an algorithm to find consistent facet orientations for simplicial meshes, which is used in FEniCS [13] and in Firedrake [20]. Other finite element packages often use different approaches. libMesh [16] uses global vertex numbers to determine the direction of edges for \(H(\text{curl})\) conforming elements, however neither \(H(\text{div})\) conforming, nor higher-order elements are supported. FreeFem++ [12] does not support quadrilaterals. Nektar++ [7] stores and uses explicit per-cell edge orientations, and also supports mixed cell meshes (e.g. mixed triangles and quadrilaterals). These orientations are employed to match degrees of freedom on facets for higher-order elements, however no sign flipping is necessary since \(H(\text{div})\) and \(H(\text{curl})\) conforming elements are not supported. Raviart-Thomas elements in DUNE [6] store explicit signs (−1 or +1) for each facet, which are referenced from manually written formulæ. deal.II [5] employs the serial algorithm we describe in §4, as a post-processing step after reading the mesh.

2. Problem Statement. This section aims to formally define the problem that we later analyse and provide algorithmic solutions to. Throughout this paper, we say that a mesh has consistent facet orientations, if:

(i) Facet orientation is intrinsic to the facet, i.e. it does not depend on the cell from which the facet is accessed.

(ii) There is a fixed reference cell such that for each cell in the mesh, there exists a mapping to the reference cell under which the orientation of each facet incident to that cell matches the orientation of the corresponding facet of the reference cell.

The orientation of facets on the reference cell can be arbitrary for the purpose of the above definition, although this choice can limit the scope of meshes for which a consistent facet orientation exists.

In this paper, we tackle the problem of finding consistent edge orientations for quadrilateral meshes on 2-dimensional orientable manifolds.

3. Proof of the Existence of a Solution. We prove that consistent edge orientations exist for any quadrilateral mesh on an orientable 2-dimensional manifold.
Later we build on the analysis below to construct algorithms that find consistent orientations.

As mentioned earlier, the choice of edge orientations on the reference quadrilateral can affect the set of meshes for which consistent orientations exist. Up to symmetry, there are four different edge orientations of a quadrilateral. These are shown in Figure 1. The symmetries arise since we can arbitrarily choose the first vertex and the direction of traversal when defining a mapping to the reference cell. For each case, one can find a mesh on a 2-dimensional manifold which does not have consistent edge orientations. [4] includes a counter-example for (a). A structured grid with odd number of cells in both direction, folded into a torus, is a counter-example for (b) and (d). A structured grid bent into a Möbius strip is a counter-example for (c). However, when we restrict our attention to meshes on orientable manifolds (that is, the vast majority of domains actually employed for finite element problems), then only (c) remains without a counter-example. We here prove existence for this case:

**Theorem 1.** Consistent edge orientations exist for any quadrilateral mesh on an orientable 2-dimensional manifold, given the edge orientations on the reference quadrilateral as in Figure 1c.

**Proof.** Notice that on the reference quadrilateral, opposite edges always have the same orientation. When constructing a global orientation of the cells in a mesh, this implies that fixing the orientation of one edge determines the orientation of the opposite edge. Conversely, the orientation of an edge imposes no restriction on the orientation of the two adjacent edges. Since each interior edge of the mesh is adjacent to two cells, fixing the orientation of one edge actually fixes the orientation of every edge reachable from the first edge by an unbroken sequence of opposite cell edges. This defines a relation “the orientation of edge a determines the orientation of edge b”. We will refer to this as the orientation determination relation. It is easy to see that this relation is reflexive, symmetric and transitive, and is therefore an equivalence relation which defines equivalence classes on the set of edges. The form of each equivalence class is a path or ribbon (see Figure 2) through the mesh connecting successive opposite faces. Within each class, the orientation of any edge determines the orientation of all other edges. Conversely, since the orientation of an edge only determines the orientation of the opposite edge and not that of the adjacent edges, separate classes can be oriented independently.

Note that if there exists a consistent edge orientation for a quadrilateral mesh, one can invert the orientation of every edge in any equivalence class, and still have a consistent edge orientation. Thus when assigning orientations to a set of dependent edges, we can arbitrarily choose the orientation of one edge, and that orientation propagates to all dependent edges. Either both initial choices are fine, or a consistent
orientation does not exist.

This implies that a consistent orientation will always exist for a mesh, unless there is a set of dependent edges for which the orientations necessarily conflict. That is, there is an edge \( u \), whose orientation forces a different orientation on an edge \( v \) through path \( p_1 \) than through path \( p_2 \). From a slightly different point of view, there is a cycle \( p = u \xrightarrow{p_1} v \xrightarrow{p_2'} \xrightarrow{p_2} u \) around \( u \), where \( p_2' \) is the reverse of \( p_2 \), such that the orientation of \( u \) forces a conflicting orientation on itself through path \( p \). Without loss of generality we can assume that \( p \) is a simple cycle, i.e. \( p \) contains each of its edges only once (except for \( u \), which appears as starting and ending edge in \( p \)). In this case the quadrilateral cells in \( p \) form a Möbius strip (see Figure 3), which implies that the mesh is not on an orientable manifold.

We have shown that if the mesh does not admit consistent edge orientations, then its manifold is non-orientable. It follows immediately that consistent orientations exist for any quadrilateral mesh on an orientable manifold. \( \square \)

4. Serial Algorithm. Based on the above discussion, Alg. 1 describes the steps of consistently orienting a quadrilateral mesh. We assume that initially \( \text{visited}[e] = \text{False} \) and \( \text{orientations}[e] \) is undefined for all \( e \in E \), where \( E \) denotes the set of edges in the mesh. For generality, lines 13 and 14 lack the details of mesh representation and orientation representation.

Lines 10–16 run only \(|E|\) times, since each time they flip a \text{False} to \text{True} in \text{visited}. Since the quadrilateral mesh is on a 2-dimensional manifold, only 1 or 2 cells can be incident to an edge (line 12), therefore lines 10–16 take \( O(|E|) \) total runtime, not considering the time spent in recursive \text{ORIENT} calls outside these lines. It also follows that \text{ORIENT} is called \( O(|E|) \) times: \(|E| \) times in line 2, and between
Figure 3: Edge orientations propagate in a conflicting manner on a Möbius strip, and vice versa, if a cycle propagates orientations in a conflicting way, then connecting the cells will form a Möbius strip.

Algorithm 1 Serial algorithm

1: for all $e \in E$ do
2:     Orient($e$, $\uparrow$)
3: end for

4: procedure Orient($e$, $o$)
5:     if visited[$e$] then
6:         if orientations[$e$] $\neq o$ then \hspace{1cm} $\triangleright$ Möbius strip found
7:             abort
8:         end if
9:     else
10:        visited[$e$] $\leftarrow$ True
11:        orientations[$e$] $\leftarrow$ $o$
12:        for all $c \in$ cells incident to $e$ do
13:            $e' \leftarrow$ edge opposite to $e$ in $c$
14:            $o' \leftarrow$ orientation for $e'$ consistent with $e$ having orientation $o$
15:            Orient($e'$, $o'$)
16:        end for
17:    end if
18: end procedure

$|E|$ and $2|E|$ times in line 15. Hence the serial algorithm takes $O(|E|)$ time.

The serial algorithm was first implemented in deal.II [5], and briefly described in its documentation [4].

5. Parallel Algorithm. This section describes the parallel extension of the above algorithm for distributed memory systems, as it is implemented in Firedrake.

The basic idea of Firedrake's parallel implementation is to first solve edge orientation locally, then flip some local ribbon segments until all local domains agree on the orientation of all shared edges (see Figure 4). To speed up flipping local ribbon
segments, each parallel process records connections between its shared edges. Interior edges are re-aligned only when agreement on the orientations of all shared edges is reached.

Alg. 2 describes how parallel processes negotiate the orientation of shared edges. Before the negotiation starts, each process aligns its edges locally, while populating the \texttt{affects\_edge} and \texttt{affects\_orient} arrays. If \( u \) and \( v \) are edges shared with other processes, and connected by a local ribbon segment, then \texttt{affects\_edge} is populated with entries \( \texttt{affects\_edge}[u] = v \) and \( \texttt{affects\_edge}[v] = u \), and \texttt{affects\_orient} is populated with entries \( \texttt{affects\_orient}[u] = \texttt{affects\_orient}[v] = \uparrow \) when \( u \) and \( v \) have the same orientation, and with entries \( \texttt{affects\_orient}[u] = \texttt{affects\_orient}[v] = \downarrow \) otherwise. If \( u \) is a shared edge, but it does not connect to any other shared edge, these arrays are not updated. If we encounter a closed loop while locally orienting edges, there is no need to negotiate the orientation its edges with other parallel processes.

Another prerequisite is to initialise the \texttt{our\_weight} and \texttt{our\_orient} arrays with the weights and proposed orientations of the shared edges of a parallel process. The weights shall be globally unique for all local ribbon segments. We use \texttt{our\_weight}[\texttt{e}] = \texttt{size} \cdot l(\texttt{e}) + \texttt{rank}, where \( l(\texttt{e}) \) is the local index of the ribbon segment which \( \texttt{e} \) belongs to, \texttt{size} is the number of parallel process, and \texttt{rank} is the index of the current process. We need to ensure that locally connected edges have the same weight and consistent orientation initially. This invariant is maintained by Alg. 2.

Alg. 2 consists of a main loop which terminates when there are no more conflicts in the proposed orientations for shared edges. In line 3–4 each parallel process exchanges its proposed orientations and weights for its shared edges with its neighbours. \texttt{their\_weight} and \texttt{their\_orient} have type and size identical to \texttt{our\_weight} and \texttt{our\_orient} respectively, but they contain the values proposed by the neighbouring processes. The general rule is to adopt the orientation with the larger weight.
Algorithm 2 Parallel algorithm

1: conflict ← True
2: while conflict do
3:   ExchangeEdgeData(our_orient, their_orient)
4:   ExchangeEdgeData(our_weight, their_weight)
5:   conflict ← False
6:   for all e ∈ shared edges do
7:     if our_weight[e] = their_weight[e] then
8:       if our_orient[e] ≠ their_orient[e] then
9:         abort
10:      end if
11:     else if our_weight[e] < their_weight[e] then
12:       our_weight[e] = their_weight[e]
13:       our_orient[e] = their_orient[e]
14:     if e ∈ affects_edge then
15:       e' ← affects_edge[e]
16:       o' ← affects_orient[e] XOR our_orient[e]
17:       if our_orient[e'] ≠ o' then
18:         conflict ← True
19:       end if
20:       our_weight[e'] = our_weight[e]
21:       our_orient[e'] = o'
22:     end if
23:   end for
24:   conflict ← AllReduce(conflict, Or)
25: end while

Lines 11–23 handle the adoption of the remote weight and orientation. Line 14 checks if such an update needs propagation via the local ribbon segment. Lines 15–21 propagate the new information to the other connected shared edge, and check if it causes that edge to flip. If yes, then another round of exchange is necessary, since the orientation for e' that this process forwarded in line 3, has now become outdated. When the local weight is larger than the remote weight, there is nothing to do: the other parallel process will adopt our orientation. Since all weights are initially unique, and then they are copied between neighbours, the local weight being equal to the remote weight means that both parallel processes have their local ribbon segment aligned to the same “source”. Therefore a mismatch in the orientation is only possible if the global mesh contains a Möbius strip, which is checked in lines 8–10.

This algorithm terminates in at most k communication rounds, where k is the maximum number of local segments in any ribbon of the global mesh. Since each ribbon is “untwisted” independently, further discussion focuses on one arbitrary ribbon in the global mesh. Any time there is a conflict between the orientation of two local segments, the one with the higher weight “wins”, thus we can conclude that the orientation of a ribbon in the global mesh is ultimately determined by the local segment which has the highest weight. Initially only one segment has this highest weight. In each communication round one or two additional segments adopt this weight and align their orientations, until the information has propagated to all segments. At that
point, there are either no conflicts along the ribbon, or the algorithm will abort in the next communication round due to finding a Möbius strip. Therefore the parallel algorithm terminates in at most \( k \) communication rounds.

Theoretically, \( k \) can be large with respect to other parameters of the mesh. For example, one can mesh an annulus domain with a very long, spiraling ribbon. However, a far more typical scenario is for each ribbon to either enter one side of the domain and exit the other, or to form a loop. In either of these cases, assuming the domain has bounded aspect ratio and that the parallel decomposition approximately minimises the surface area of partitions, \( k \) will be \( O(\sqrt{P}) \), where \( P \) is the number of parallel processes.

6. Experiments. We have run several scaling experiments on the ARCHER UK National Supercomputing Service. Our experimentation framework is available as [14]. We used 6 different meshes to evaluate the performance of the parallel algorithm described in §5:

- \texttt{s.square}: structured grid on a square domain
- \texttt{s.sphere}: cubed sphere mesh (a quadrilateral mesh of the surface of a sphere formed by deforming a cube to sphere shape and refining)
- \texttt{u.square}: unstructured mesh on a square domain
- \texttt{u.sphere}: unstructured mesh on the surface of a sphere
- \texttt{t10}, \texttt{t11}: unstructured meshes with non-uniform resolution (see Figure 5)

The first two were generated using Firedrake utility functions, the other four were generated with Gmsh [10], version 2.8.3. \texttt{t10} and \texttt{t11} are examples from the Gmsh tutorial. To give an impression of the level of irregularity of these meshes, low-resolution versions are presented in Figure 5. We used meshes with cell count in the order of millions, unfortunately mesh generation becomes very difficult for meshes bigger than that.

We used Firedrake to construct consistent edge orientations for each of these meshes on up to 24576 cores (1024 computing nodes, each with 24 CPU cores). We measured the number of required communication rounds (Figure 6) and execution time (Figure 7). Each experiment was repeated 4 times, except for those involving the largest number of cores: for cost efficiency reasons, they were carried out only once. The number of communication rounds were the same during each instantiation of the same experiment, however, the execution times varied greatly, especially when they were under a second. Therefore reported execution times should be taken with a grain of salt. Figure 7 shows the average of 4 measurements for each data point.
Unsurprisingly, the construction edge orientations for structured meshes takes relatively few communication rounds. Figure 6 also shows that for a rectangular domain, the number of required communication rounds stays the same order of magnitude in case of non-uniform meshing as with uniform resolution. However, the unstructured mesh on a spherical domain requires at least one order of magnitude more communication rounds. This is probably caused by the fact that a sphere does not have a boundary, thus all ribbons must form a closed loop, so traversing a ribbon must continue until the starting edge is reached again. Regardless, our measurements confirm
Table 1: Slopes of lines fitted to the log-log data shown in Figure 6.

| Test case | Slope  |
|-----------|--------|
| u_square  | 0.485698 |
| s_square  | 0.527252 |
| t10       | 0.470928 |
| t11       | 0.494417 |
| s_sphere  | 0.446810 |
| u_sphere  | 0.526821 |

In most of the cases shown in Figure 7, the execution time initially decreases as the number of parallel processes increases. It is likely that this is due to the parallelisation of the local work each process has to do. Eventually the increasing number of communication rounds becomes dominant, and the execution times grow with the number parallel processes. However, even the unstructured sphere mesh with almost one million cells, running on about a quarter of ARCHER, takes only 105 seconds. For users running supercomputer-scale simulations, the few minutes spent on edge orientations are not likely to become a bottleneck.

7. Alternative Approaches to Parallelism. The edge orientation problem can be reduced to the well-known Union-Find algorithm, also known as the disjoint set data structure [8]. We established in §3 that the orientation determination relation is an equivalence relation which defines equivalence classes (the “ribbons”) on the set of edges. To discover the ribbons, we can immediately use the Union-Find algorithm. However, with an extension to it, we can also retrieve the correct edge orientations.

First we briefly introduce the Union-Find algorithm. Let $e_1, e_2, \ldots, e_n$ be abstract elements, and $S_1, S_2, \ldots, S_k$ disjoint non-empty sets over these elements, with $k \leq n$. The sets are initially singletons: $S_i = \{ e_i \}, i = 1 \ldots n$. The UNION operation destructively joins two sets. The FIND operation can be used to determine whether two elements, $e_i$ and $e_j$, are in the same set. Since the sets are disjoint, each element is in exactly one set. Each set is identified by its representative element, which can be any of its elements. The FIND operation returns, for any element $e_i$, the representative element of the set to which $e_i$ belongs. Thus $\text{FIND}(e_i) = \text{FIND}(e_j)$ if $e_i$ and $e_j$ are in the same set.

To solve the edge orientation problem, the edges of the mesh become the abstract elements for the Union-Find algorithm, starting with a singleton set for each edge. Then, for each cell, we join both pairs of opposite edges, more precisely we join the sets to which those opposite edges belong. When done, each set corresponds to a “ribbon”, a set of edges which determine each other’s orientation. However, we still need to traverse the ribbon to determine the permitted relative orientations between edges. We will avoid this traversal by extending the Union-Find algorithm, but let us first briefly present the usual representation of sets and the algorithms for FIND and UNION.

Each element $u$ links to its parent element $p(u)$. For representative elements, $p(u) = u$. For all other elements, $p(u)$ is another element in the same set, and following the parent links must eventually lead to the representative element of the
set. Alg. 3 and Alg. 4 show the implementation of the Find and Union operations, respectively.

**Algorithm 3** Find operation for disjoint sets

1: function Find(u)
2: if \( p(u) = u \) then
3: return \( u \)
4: else
5: return Find(p(u))
6: end if
7: end function

**Algorithm 4** Union operation for disjoint sets

1: procedure Union(u, v)
2: \( r_u \) ← Find(u)
3: \( r_v \) ← Find(v)
4: if \( r_u \neq r_v \) then \( \triangleright r_v \) becomes the representative element of the united set
5: \( p(r_u) \) ← \( r_v \)
6: end if
7: end procedure

Find simply traverses the parent links until it reaches the representative element. A variation of Find also reassigns the parent links of each element on this path directly to the representative element — this technique is called **path compression**. Union first calls Find to look up the representative elements of both sets. If the two sets are indeed different, then it assigns the parent link of one of the representative elements to point to the other representative element, thus effectively merging the sets. Depending on the exact variant of the Union-Find algorithm [18], there is generally a rule to determine which one of \( r_u \) and \( r_v \) remains a representative element, but we ignore that detail for this discussion. The textbook version of the Union-Find algorithm [22] uses rank based merging in Union and path compression in Find, and it is proven that \( m \) Find and \( n \) Union operations execute in \( O(m + n)\alpha(m, n) \) time, where \( \alpha(m, n) \), the functional inverse of Ackermann’s function, is so slow-growing that for all practical purposes it can be considered constant.

To address the edge orientation problem, we attach a binary value \( \omega(u) \) to each parent link, which describes the permitted relative orientation between the edge \( u \) and its parent edge \( p(u) \). Alg. 5 and Alg. 6 describe Find and Union with orientations, respectively. The changes include the maintenance of orientation, and Möbius strip detection in Union when one tries to connect a ribbon to itself with the inconsistent orientation.

This extension can be easily applied to most of the relevant variants of the Union-Find algorithm, including parallel ones. Alg. 7 describes the algorithm to solve the edge orientation problem using the Union-Find algorithm extended with orientations. While Alg. 7 itself does not feature any parallel primitives, it will solve the edge orientation problem in parallel if a parallel variant of the Union-Find algorithm is employed. In that case each parallel process only iterates over its local cells and edges.
Algorithm 5 \textbf{FIND} operation for disjoint sets \textit{with orientation}

1: function FIND\((u)\)
2: \quad if \(p(u) = u\) then
3: \quad \quad return \(u, \uparrow\)
4: \quad else
5: \quad \quad \(r_u, o_p \leftarrow\) FIND\((p(u))\)
6: \quad \quad return \(r_u, \omega(u) \text{ XOR } o_p\)
7: \quad end if
8: end function

Algorithm 6 \textbf{UNION} operation for disjoint sets \textit{with orientation}

1: procedure UNION\((u, v, o)\)
2: \quad \(r_u, o_u \leftarrow\) FIND\((u)\)
3: \quad \(r_v, o_v \leftarrow\) FIND\((v)\)
4: \quad if \(r_u = r_v\) then
5: \quad \quad if \(o_u \text{ XOR } o_v = \downarrow\) then \(\triangleright\) Möbius strip found
6: \quad \quad \quad abort
7: \quad \quad end if
8: \quad else \(\triangleright r_v\) becomes the representative element of the united set
9: \quad \quad \(p(r_u) \leftarrow (r_v, o_u \text{ XOR } o_v)\)
10: \quad end if
11: end procedure

The advantage of this approach is that we can easily re-use existing work on parallelising the Union-Find algorithm to solve the edge orientation problem. Several attempts have been made to parallelise that algorithm both on shared memory computers [9, 2, 3, 19], and on distributed memory systems [9, 17, 11, 15]. Since Firedrake is meant to scale on today’s supercomputers, we are only interested in distributed memory parallelism. Cybenko et al. [9] attempt both shared memory and distributed memory parallelisation, but their distributed memory algorithm suffered from slowdown with increasing number of processors and constant problem size. Manne and Patwary [17] propose a distributed memory parallel algorithm, which they demonstrate to scale up to 40 processors for certain graphs. This is, however, far from the scale of supercomputers on which Firedrake is supposed to run. Harrison et al. [11] achieve good scaling for their purposes, which is connected component labeling for distributed memory visualisation tools. However, the final step of their algorithm uses all-to-all communication to merge the local components, which we expect to become a serious bottleneck for our purposes, since distributed quadrilateral meshes produce numerous local ribbon segments and connections between them. Iverson et al. [15] compare five parallel algorithms for connected component labeling on distributed memory systems, both theoretically and experimentally. The approach they call label propagation is essentially equivalent to the parallelisation of the edge orientation algorithm implemented in Firedrake. Although it was not always the fastest in their evaluations, it did not suffer from the explosive use of memory which affected some other algorithms.

8. Conclusion & Outlook. We have proposed a distributed mesh parallel algorithm, extending the serial quadrilateral edge orientation algorithm in deal.II. We
Algorithm 7 Edge orientations using the extended Union-Find algorithm

| Line | Description |
|------|-------------|
| 1:   | for all $c \in \text{local}$ cells do |
| 2:   | $e_1, e_2, e'_1, e'_2 \leftarrow$ edges of $c$, each consecutive pair sharing a vertex |
| 3:   | $o_1 \leftarrow$ permitted relative orientation between $e_1$ and $e'_1$ |
| 4:   | $o_2 \leftarrow$ permitted relative orientation between $e_2$ and $e'_2$ |
| 5:   | $\text{Union}(e_1, e'_1, o_1)$ |
| 6:   | $\text{Union}(e_2, e'_2, o_2)$ |
| 7:   | end for |
| 8:   | for all $e \in \text{local}$ edges do |
| 9:   | $r_e, o_e \leftarrow \text{Find}(e)$ |
| 10:  | $\text{orientations}[e] \leftarrow o_e$ |
| 11:  | end for |

have proven that there are consistent edge orientations for any quadrilateral mesh of an orientable 2-dimensional manifold. Our novel analysis establishes a link between the well-known Union-Find algorithm and assigning orientations to the edges of a quadrilateral mesh, thus existing parallelisation approaches for the Union-Find algorithm can be easily adapted to construct further parallel algorithms for mesh orientations.

**REFERENCES**

[1] Rainer Agelek, Michael Anderson, Wolfgang Bangerth, and William Barth, On orienting edges of unstructured two- and three-dimensional meshes, arXiv, (2015).
[2] Richard J. Anderson and Heather Woll, Wait-free parallel algorithms for the union-find problem, in Proceedings of the Twenty-third Annual ACM Symposium on Theory of computing - STOC ’91, 1991, pp. 370–380.
[3] David A. Bader and Guojing Cong, A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs), Journal of Parallel and Distributed Computing, 65 (2005), pp. 994–1006.
[4] Wolfgang Bangerth, Michael Anderson, and Ralf Hartmann, The deal.II Library: GridReordering<dim, spacedim> Class Template Reference. https://www.dealii.org/developer/doxygen/deal.II/classGridReordering.html. Accessed: 2015-04-08.
[5] W. Bangerth, R. Hartmann, and G. Kanschat, deal.II – A general-purpose object-oriented finite element library, ACM Transactions on Mathematical Software, 33 (2007), pp. 24/1–24/7.
[6] Peter Bastian, Markus Blatt, Christian Engwer, Andreas Dedner, Robert Klöfkorn, Sreenadh P. Kuttenikkad, Mario Ohlberger, and Oliver Sander, The Distributed and Unified Numerics Environment (DUNE), in Proceedings of the 19th Symposium on Simulation Technique in Hannover, 2006.
[7] C. D. Cantwell, D. Mony, A. Comerford, A. Bols, G. Rocco, G. Mengaldo, D. de Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelischot, B. Jordi, H. Xu, Y. Mohamed, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin, Nektar++: An open-source spectral/hp element framework, Computer Physics Communications, (2015).
[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Others, Introduction to algorithms, vol. 2, MIT Press Cambridge, 2001.
[9] G. Cybenko, T. G. Allen, and J. E. Polito, Practical parallel Union-Find algorithms for transitive closure and clustering, International Journal of Parallel Programming, 17 (1988), pp. 403–423.
[10] Christophe Geuzaine and Jean-François Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, International Journal for Numerical Methods in Engineering, 79 (2009), pp. 1309–1331.
[11] Cyrus Harrison, Hank Childs, and Kelly P. Gaither, Data-parallel mesh connected com-
ponents labeling and analysis, in EG PGV’11 Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization, 2011, pp. 131–140.

[12] F Hecht, New development in FreeFem++, Journal of Numerical Mathematics, 20 (2012), pp. 251–265.

[13] J. Hoffman, J. Jansson, C. Johnson, M. G. Knepley, R. C. Kirby, A. Logg, L. R. Scott, and G. N. Wells, FEniCS. http://fenicsproject.org/, 2006. Accessed: 2015-04-08.

[14] Miklós Homolya, Experimentation framework for manuscript “A parallel edge orientation algorithm for quadrilateral meshes”, version 2. http://dx.doi.org/10.5281/zenodo.31190, Sept. 2015.

[15] J. Iverson, C. Kamath, and G. Karypis, Evaluation of connected-component labeling algorithms for distributed-memory systems, Parallel Computing, 44 (2015), pp. 53 – 68.

[16] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey, libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations, Engineering with Computers, 22 (2006), pp. 237–254.

[17] Fredrik Manne and Md. Mostofa Ali Patwary, A scalable parallel union-find algorithm for distributed memory computers, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6067 LNCS (2010), pp. 186–195.

[18] Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne, Experiments on union-find algorithms for the disjoint-set data structure, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6049 LNCS (2010), pp. 411–423.

[19] Md. Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne, Multi-core spanning forest algorithms using the disjoint-set data structure, Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012, (2012), pp. 827–835.

[20] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T T McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul H J Kelly, Firedrake: automating the finite element method by composing abstractions, Submitted to ACM TOMS, (2015).

[21] Marie Rognes, Robert C Kirby, and Anders Logg, Efficient Assembly of H(div) and H(curl) Conforming Finite Elements, SIAM Journal on Scientific Computing, 31 (2012), pp. 4130–4151.

[22] Robert Endre Tarjan, Efficiency of a good but not linear set union algorithm, Journal of the ACM (JACM), 22 (1975), pp. 215–225.