Thread-Parallel Anisotropic Mesh Adaptation

Gerard J. Gorman, Georgios Rokos, James Southern, and Paul H.J. Kelly

Abstract Mesh adaptation is a powerful way to minimise the computational cost of mesh based computation. It is particularly successful for multi-scale problems where the required mesh resolution can vary by orders of magnitude across the domain. The end result is local control over solution accuracy and reduced time to solution.

In the case of large scale simulations, where the time to solution is unacceptable or the memory requirements exceeds available RAM, mesh based computation is typically parallelised using domain decomposition methods using the Message Passing Interface (MPI). This allows a simulation to run in parallel on a distributed memory computer. While this has been a high successful strategy up until now, the drive towards low power multi- and many-core architectures means that an even higher degree of parallelism is required and the memory hierarchy exploited to maximise memory bandwidth.

For this reason application codes are increasingly adopting a hybrid parallel approach whereby decomposition methods, implemented using the Message Passing Interface (MPI), are applied for inter-node parallelisation, while a threaded programming model is used for intra-node parallelisation. Mesh adaptivity has been successfully parallelised using MPI by a number of groups, and can be implemented efficiently with few modifications to the serial code. However, thread-level parallelism is significantly more challenging because each thread modifies the mesh data and therefore must be carefully marshalled to avoid data races while still ensuring enough parallelism is exposed to achieve good parallel efficiency.

G.J. Gorman (✉)
Department of Earth Science and Engineering, Imperial College London, London SW8 2AZ, UK
e-mail: g.gorman@imperial.ac.uk

G. Rokos • P.H.J. Kelly
Department of Computing, Imperial College London, London SW7 2AZ, UK
e-mail: georgios.rokos09@imperial.ac.uk; p.kelly@imperial.ac.uk

J. Southern
Fujitsu Laboratories of Europe, Hayes Park Central, Hayes End Road, Hayes,
Middlesex UB4 8FE, UK
e-mail: James.Southern@uk.fujitsu.com

S. Perotto, L. Formaggia (eds.), *New Challenges in Grid Generation and Adaptivity for Scientific Computing*, SEMA SIMAI Springer Series 5,
DOI 10.1007/978-3-319-06053-8_6

Here we describe a new thread-parallel algorithm for anisotropic mesh adaptation algorithms. For each mesh optimisation phase (refinement, coarsening, swapping and smoothing) we describe how independent sets of tasks are defined. We show how a deferred updates strategy can be used to update the mesh data structures in parallel and without data contention. We show that despite the complex nature of mesh adaptation and inherent load imbalances in the mesh adaptivity, good parallel efficiency can be achieved.

1 Introduction

Anisotropic mesh adaptation methods provide an important means to minimise superfluous computation associated with over-resolving the solution while still achieving the required accuracy, [1, 5, 17, 18]. In order to use mesh adaptation within a simulation, the application code requires a method to estimate the local solution error. Given an error estimate it is then possible to calculate the required local mesh resolution in order to achieve a specific solution accuracy.

There are a number of examples where adaptive mesh methods have been parallelised in the context of distributed memory parallel computers. The main challenge in performing mesh adaptation in parallel is maintaining a consistent mesh across domain boundaries. One approach is to first lock the regions of the mesh which are shared between processes and for each process to apply the serial mesh adaptation operation to the rest of the local domain. The domain boundaries are then perturbed away from the locked region and the lock-adapt iteration is repeated [7]. Freitag et al. [12, 13] considers a fine grained approach whereby a global task graph is defined which captures the data dependencies for a particular mesh adaptation kernel. This graph is coloured in order to identify independent sets of operations. The parallel algorithm then progresses by selecting an independent set (vertices of the same colour) and applying mesh adaptation kernels to each element of the set. Once a sweep through a set has been completed, data is synchronised between processes, and a new independent set is selected for processing. In [3] each process applies the serial adaptive algorithm, however rather than locking the halo region, operations to be performed on the halo are first stashed in buffers and then communicated so that the same operations will be performed by all processes that share mesh information. For example, when coarsening is applied all the vertices to be removed are computed. All operations which are local are then performed while pending operations in the shared region are exchanged. Finally, the pending operations in the shared region are applied.

However, over the past decade there has been a trend towards multi- and many-core compute nodes. Indeed, it is assumed that the compute nodes of a future exascale supercomputer will each contain thousands or even tens of thousands of cores [9]. On multi-core architectures, a popular parallel programming paradigm is to use thread-based parallelism to exploit shared memory within a shared memory node and a message passing using MPI for interprocess communication. When the computational intensity is sufficiently high, a third level of parallelisation may be

implemented via SIMD instructions at the core level. There are some opportunities to improve performance and scalability by reducing communication needs, memory consumption, resource sharing as well as improved load balancing [19]. However, the algorithms themselves must also have a high degree of thread parallelism if they are to have a future on multi-core architectures; whether it be CPU or coprocessor based.

Rokos et al. [20] and Gorman et al. [16] develop thread parallel algorithms for many-core and multi-core processors based on the independent set approach described in [12]. However, this approach does not easy carry over for a threaded implementation of the other mesh adaptation operations. Therefore, in this paper we take a fresh look at the anisotropic adaptive mesh methods in 2D to develop new scalable thread-parallel algorithms suitable for modern multi-core architectures. We show that despite the irregular data access patterns, irregular workload and need to rewrite the mesh data structures, good parallel efficiency can be achieved.

The algorithms described in this paper have been implemented in the open source code PRAgMaTIc (Parallel anisotRopic Adaptive Mesh ToolkIt).[1] The remainder of the paper is laid out as follows: Sect. 2 gives an overview of the anisotropic adaptive mesh procedure; Sect. 3 describes the thread-parallel algorithm; and Sect. 4 illustrates how well the algorithm scales for a benchmark problem. We conclude with a discussion on future work and possible implications of this work.

2 Overview

In this section we will give an overview of anisotropic mesh adaptation. In particular, we focus on the element quality as defined by an error metric and the anisotropic adaptation kernels which iteratively improve the local mesh quality as measured by the worst local element.

2.1 Error Control

Solution discretisation errors are closely related to the size and the shape of the elements. However, in general meshes are generated using *a priori* information about the problem under consideration when the solution error estimates are not yet available. This may be problematic because:

- Solution errors may be unacceptably high.
- Parts of the solution may be over-resolved, thereby incurring unnecessary computational expense.

[1] https://code.launchpad.net/~pragmatic-core/pragmatic/pragmatic2d-2.0

A solution to this is to compute appropriate local error estimates, and use this information to compute a field on the mesh which specifies the local mesh resolution requirement. In the most general case this is a metric tensor field so that the resolution requirements can be specified anisotropically; for a review of the procedure see [14]. Size gradation control can be applied to this metric tensor field to ensure that there are not abrupt changes in element size [2].

2.2 Element Quality

As discretisation errors are dependent upon element shape as well as size, a number of different measures of element quality have been proposed, e.g. [1, 5, 18, 21, 22]. Here we use the element quality measure for triangles proposed by [22]:

$$q^M(\triangle) = \underbrace{12\sqrt{3}\frac{|\triangle|_M}{|\partial\triangle|_M^2}}_{I} \underbrace{F\left(\frac{|\partial\triangle|_M}{3}\right)}_{II}, \tag{1}$$

where $|\triangle|_M$ is the area of element $\triangle$ and $|\partial\triangle|_M$ is its perimeter as measured in a Riemannian space locally defined by the metric tensor M as evaluated at the element's centre. The first factor (I) is used to control the shape of element $\triangle$. For an equilateral triangle with sides of length l in metric space, $|\triangle| = l^2\sqrt{3}/4$ and $|\partial\triangle| = 3l$; and so $I = 1$. For non-equilateral triangles, $I < 1$. The second factor (II) controls the size of element $\triangle$. The function F is smooth and defined as:

$$F(x) = (\min(x, 1/x)(2 - \min(x, 1/x)))^3, \tag{2}$$

which has a single maximum of unity with $x = 1$ and decreases smoothly away from this with $F(0) = F(\infty) = 0$. Therefore, $II = 1$ when the sum of the lengths of the edges of $\triangle$ is equal to 3, i.e. an equilateral triangle with sides of unit length, and $II < 1$ otherwise. Hence, taken together, the two factors in (1) yield a maximum value of unity for an equilateral triangle with edges of unit length, and decreases smoothly to zero as the element becomes less ideal.

2.3 Overall Adaptation Procedure

The mesh is adapted through a series of local operations: edge collapse (Sect. 2.4.1); edge refinement (Sect. 2.4.2); element-edge swaps (Sect. 2.4.3); and vertex smoothing (Sect. 2.4.4). While the first two of these operations control the local resolution, the latter two operations are used to improve the element quality.

Algorithm 1 Mesh optimisation procedure

Inputs: $\mathscr{M}$, $\mathscr{S}$.
$(\mathscr{M}^*, \mathscr{S}^*) \leftarrow coarsen(\mathscr{M}, \mathscr{S})$
repeat
  $(\mathscr{M}^*, \mathscr{S}^*) \leftarrow refine(\mathscr{M}^*, \mathscr{S}^*)$
  $(\mathscr{M}^*, \mathscr{S}^*) \leftarrow coarsen(\mathscr{M}^*, \mathscr{S}^*)$
  $(\mathscr{M}^*, \mathscr{S}^*) \leftarrow swap(\mathscr{M}^*, \mathscr{S}^*)$
until (maximum number of iterations reached) **or** (algorithm convergence)
$(\mathscr{M}^*, \mathscr{S}^*) \leftarrow smooth(\mathscr{M}^*, \mathscr{S}^*)$
return $\mathscr{M}^*$

Algorithm 1 gives a high level view of the anisotropic mesh adaptation procedure as described by Li et al. [17]. The inputs are $\mathscr{M}$, the piecewise linear mesh from the modelling software, and $\mathscr{S}$, the node-wise metric tensor field which specifies anisotropically the local mesh resolution requirements. Coarsening is initially applied to reduce the subsequent computational and communication overheads. The second stage involves the iterative application of refinement, coarsening and swapping to optimise the resolution and the quality of the mesh. The algorithm terminates once the mesh optimisation algorithm converges or after a maximum number of iterations has been reached. Finally, mesh quality is fine-tuned using some vertex smoothing algorithm (e.g. quality-constrained Laplacian smoothing [11], optimisation-based smoothing [12]), which aims primarily at improving worst-element quality.

2.4 Adaptation Kernels

2.4.1 Coarsening

Coarsening is the process of lowering mesh resolution locally by removing mesh elements, leading to a reduction in the computational cost. Here this is done by collapsing an edge to a single vertex, thereby removing all elements that contain this edge. An example of this operation is shown in Fig. 1.

2.4.2 Refinement

Refinement is the process of increasing mesh resolution locally. It encompasses two operations: splitting of edges; and subsequent division of elements. When an edge is longer than desired, it is bisected. An element can be split in three different ways, depending on how many of its edges are bisected:

1. When only one edge is marked for refinement, the element can be split across the line connecting the mid-point of the marked edge and the opposite vertex. This

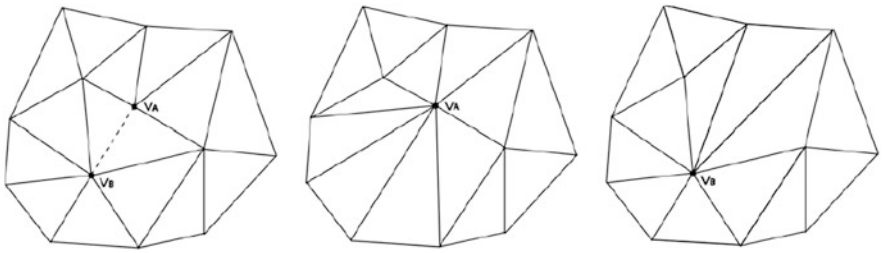

Fig. 1 Edge collapse: the dashed edge in the *left figure* is reduced into a single vertex by bringing vertex V_B on top of vertex V_A, as can be seen in the *middle figure*. The two elements that used to share the dashed edge are deleted. Edge collapse is an oriented operation, i.e. eliminating the edge by moving V_B onto V_A results in a different local patch than moving V_A onto V_B, which can be seen in the *right figure*

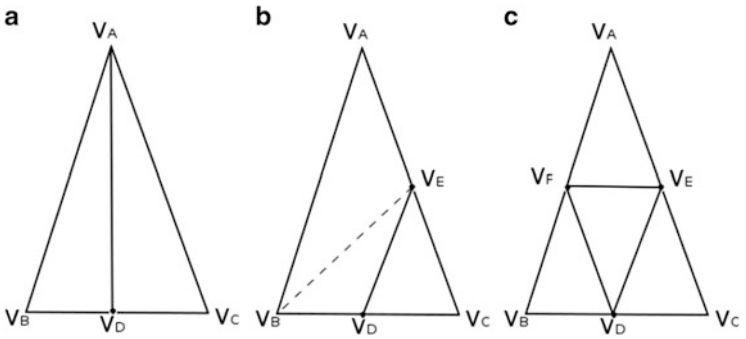

Fig. 2 Mesh resolution can be increased either by bisecting an element across one of its edges (1:2 split, **a**), by performing a 1:3 split (**b**) or by performing regular refinement to that element (1:4 split, **c**)

operation is called bisection and an example of it can be seen on the left side of Fig. 2 (left shape).

2. When two edges are marked for refinement, the element is divided into three new elements. This case is shown in Fig. 2 (middle shape). The parent element is split by creating a new edge connecting the mid-points of the two marked edges. This leads to a newly created triangle and a non-conforming quadrilateral. The quadrilateral can be split into two conforming triangles by dividing it across one of its diagonals, whichever is shorter.
3. When all three edges are marked for refinement, the element is divided into four new elements by connecting the mid-points of its edges with each other. This operation is called regular refinement and an example of it can be seen in Fig. 2 (right shape).

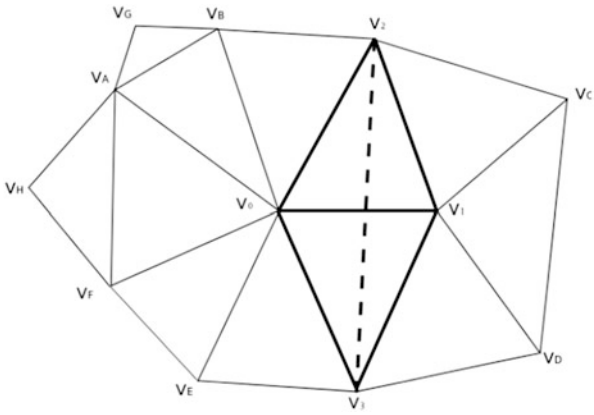

Fig. 3 Flipping the common edge $\overline{V_0V_1}$ results in the removal of triangles $\widehat{V_0V_1V_2}$ and $\widehat{V_0V_1V_3}$ and their replacement with new triangles $\widehat{V_0V_2V_3}$ and $\widehat{V_1V_2V_3}$

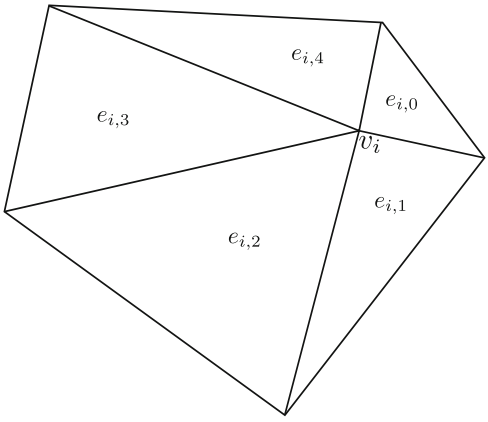

Fig. 4 Local mesh patch: $\mathbf{v}_i$ is the vertex being relocated; $\{e_{i,0}, \ldots, e_{i,m}\}$ is the set of elements connected to $\mathbf{v}_i$

2.4.3 Swapping

In 2D, swapping is done in the form of edge flipping, i.e. flipping an edge shared by two elements, see Fig. 3. The operation considers the quality of the swapped elements—if the minimum element quality has improved then the original mesh triangles are replaced with the edge swapped elements.

2.4.4 Quality Constrained Laplacian Smooth

The kernel of the vertex smoothing algorithm should relocate the central vertex such that the local mesh quality is increased (see Fig. 4). Probably the best known heuristic for mesh smoothing is Laplacian smoothing, first proposed by Field [10]. This method operates by moving a vertex to the barycentre of the set of vertices connected by a mesh edge to the vertex being repositioned. The same approach can be implemented for non-Euclidean spaces; in that case all measurements of

Algorithm 2 Smart smoothing kernel: a Laplacian smooth operation is accepted only if it does not reduce the infinity norm of local element quality

$\mathbf{v}_i^0 \leftarrow \mathbf{v}_i$
$quality^0 \leftarrow Q(\mathbf{v}_i)$
$n \leftarrow 1$
$\mathbf{v}_i^n \leftarrow \mathbf{v}_i^{\mathscr{L}}$ ▷ Initialise vertex location using Laplacian smooth
$M_i^n \leftarrow metric\_interpolation(\mathbf{v}_i^n)$
$quality^n = Q(\mathbf{v}_i^n)$ ▷ Calculate the new local quality for this relocation.
while $(n \leq max\_iteration)$**and**$(quality_i^n - quality_i^0 < \sigma_q)$ **do**
  $\mathbf{v}_i^{n+1} \leftarrow (\mathbf{v}_i^n + \mathbf{v}_i^0)/2$
  $M_i^{n+1} \leftarrow metric\_interpolation(\mathbf{v}_i^{n+1})$
  $quality^{n+1} \leftarrow Q(\mathbf{v}_i^{n+1})$
  $n = n + 1$
if $quality_i^n - quality_i^0 > \sigma_q$ **then** ▷ Accept if local quality is improved
  $\mathbf{v}_i \leftarrow \mathbf{v}_i^n$
  $\mathbf{M_i} \leftarrow \mathbf{M_i^n}$

length and angle are performed with respect to a metric tensor field that describes the desired size and orientation of mesh elements [18]. Therefore, in general, the proposed new position of a vertex $\mathbf{v}_i^{\mathscr{L}}$ is given by

$$\mathbf{v}_i^{\mathscr{L}} = \frac{\sum_{j=1}^{J} ||\mathbf{v}_i - \mathbf{v}_j||_M \mathbf{v}_j}{\sum_{j=1}^{J} ||\mathbf{v}_i - \mathbf{v}_j||_M}, \tag{3}$$

where $\mathbf{v}_j$, $j = 1, \ldots, J$, are the vertices connected to $\mathbf{v}_i$ by an edge of the mesh, and $||\cdot||_M$ is the norm defined by the edge-centred metric tensor M_{ij}. In Euclidean space, M_{ij} is the identity matrix.

As noted by Field [10], the application of pure Laplacian smoothing can actually decrease local element quality; at times, elements can even become inverted. Therefore, repositioning is generally constrained in some way to prevent local decreases in mesh quality. One variant of this, termed *smart Laplacian smoothing* by Freitag and Ollivier-Gooch [11] (while they only consider the Euclidean geometry it is straightforward to extend to Riemannian geometry), is summarised in Algorithm 2. This method accepts the new position defined by a Laplacian smooth only if it increases the infinity norm of local element quality, Q_i (i.e. the quality of the worst local element):

$$Q(\mathbf{v}_i) \equiv \|\boldsymbol{q}\|_\infty, \tag{4}$$

where i is the index of the vertex under consideration and $\boldsymbol{q}$ is the vector of the element qualities from the local patch.

3 Thread-Level Parallelism in Mesh Optimisation

To allow fine grained parallelisation of anisotropic mesh adaptation we make extensive use of maximal independent sets. This approach was first suggested in a parallel framework proposed by Freitag et al. [13]. However, while this approach avoids updates being applied concurrently to the same neighbourhood, data writes will still incur significant lock and synchronisation overheads. For this reason we incorporate a deferred updates strategy, described below, to minimise synchronisations and allow parallel writes.

In the same paper [13] the authors describe the need for propagation of operations. Adaptive operations need to be propagated to adjacent vertices/edges because topological changes or changes in element quality might give rise to new configurations of better quality.

3.1 Design Choices

Before presenting the adaptive algorithms, it is necessary to give a brief description of the data structures used to store mesh-related information. Following that, we present a set of techniques which help us avoid hazards and data races and guarantee fast and safe concurrent read/write access to mesh data.

3.1.1 Mesh Data Structures

The minimal information necessary to represent a mesh is an element-node list (we refer to it in this article as `ENList`), which is implemented in PRAgMaTIc as a C++ Standard template library (STL) vector container class storing vertex IDs (`std::vector<int>`), and an array of vertex coordinates (referred to as `coords`), which is an STL vector of coordinates (`std::vector<double>`). Element `eid` is comprised of vertices `ENList[3*eid]`, `ENList[3*eid+1]` and `ENList[3*eid+2]`, whereas the x- and y-coordinates of vertex `vid` are stored in `coords[2*vid]` and `coords[2*vid+1]` respectively. The metric tensor field is similarly stored in the STL vector `metric`.

All necessary structural information about the mesh can be extracted from `ENList`. However, it is convenient to create and maintain two additional adjacency-related structures, the node-node adjacency list (referred to as `NNList`) and the node-element adjacency list (referred to as `NEList`). As `NNList` is a ragged array it is implemented as `std::vector< std::vector<int> >` where the vector `NNList[vid]` contains the IDs of all vertices adjacent to vertex `vid`. Similarly, `NEList` is implemented as an STL vector of STL sets of element IDs (`std::vector< std::set<int> >`) and `NEList[vid]` contains the IDs of all elements which vertex `vid` is part of.

It should be noted that, contrary to common approaches in other adaptive frameworks, we do not use other adjacency-related structures such as element-element or edge-edge lists. Manipulating these lists and maintaining their consistency throughout the adaptation process is quite complex and constitutes an additional parallel overhead. Instead, we opted for the approach of finding all necessary adjacency information on the fly using `ENList`, `NNList` and `NEList`.

3.1.2 Colouring

There are two types of hazards when running mesh optimisation algorithms in parallel: structural hazards and data races. The term *structural hazards* refers to the situation where an adaptive operation results in invalid or non-conforming edges and elements. For example, on the local patch in Fig. 3, if two threads flip edges $\overline{V_0V_1}$ and $\overline{V_0V_2}$ at the same time, the result will be two new edges $\overline{V_2V_3}$ and $\overline{V_1V_B}$ crossing each other. Structural hazards for all adaptive algorithms are avoided by colouring a graph whose nodes are defined by the mesh vertices and edges are defined by the mesh edges. Maximal independent sets are readily selected by calculating the intersection between the set of vertices of each colour and the set of active vertices.

The fact that topological changes are made to the mesh means that after an independent set has been processed the graph colouring has to be recalculated. Therefore, a fast scalable graph colouring algorithm is vital to the overall performance. In this work we use a parallel colouring algorithm described by [15]. This algorithm can be described as having three stages: (a) initial pseudo-colouring where vertices are coloured in parallel and invalid colourings are possible; (b) loop over the graph to detect invalid colours arising from the first stage; (c) the detected invalid colours are resolved in serial. Between adaptive sweeps through independent sets it is only necessary to execute stages (b) and (c) to resolve the colour conflicts introduced by changes to the mesh topology.

3.1.3 Deferred Operations Mechanism

Data race conditions can appear when two or more threads try to update the same adjacency list. An example can be seen in Fig. 5. Having coloured the mesh, two threads are allowed to process vertices V_B and V_C at the same time without structural hazards. However, `NNList[` V_A `]` and `NEList[` V_A `]` must be updated. If both threads try to update them at the same time there will be a data race which could lead to data corruption. One solution could be a distance-2 colouring of the mesh (a distance-k colouring of $\mathscr{G}$ is a colouring in which no two vertices share the same colour if these vertices are up to k edges away from each other or, in other words, if there is a path of length $\leq k$ from one vertex to the other). Although this solution guarantees a race-free execution, a distance-2 colouring would increase the chromatic number, thereby reducing the size of the independent sets and therefore the available parallelism. Therefore, an alternative solution is sought.

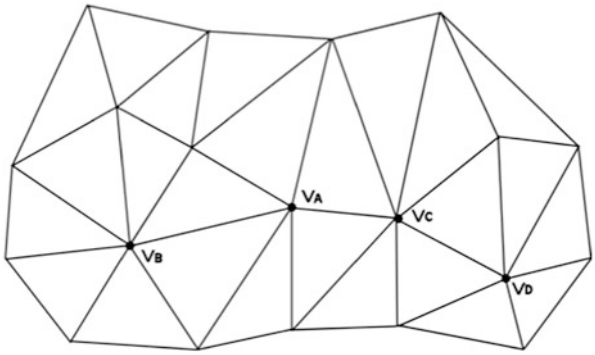

Fig. 5 Example of hazards when running edge collapse in parallel. V_B is about to collapse onto V_A. The operation is executed by thread T_1. Clearly, V_A cannot collapse at the same time. Additionally, V_C cannot collapse either, because it affects V_A's adjacency list. If a thread T_2 executes the collapse operation collapse on V_C, then both T_1 and T_2 will attempt to modify V_A's adjacency list concurrently, which can lead to data corruption. This race can be eliminated using the deferred-updates mechanism

In a shared-memory environment with `nthreads` OpenMP threads, every thread has a private collection of `nthreads` lists, one list for each OpenMP thread. When `NNList[i]` or `NEList[i]` have to be updated, the thread does not commit the update immediately; instead, it pushes the update back into the list corresponding to thread with ID $tid = i\%nthreads$. At the end of the adaptive algorithm, every thread `tid` visits the private collections of all OpenMP threads (including its own), locates the list that was reserved for `tid` and commits the operations which are stored there. This way, it is guaranteed that for any vertex V_i, `NNList[` V_i `]` and `NEList[` V_i `]` will be updated by only one thread. Because updates are not committed immediately but are deferred until the end of the iteration of an adaptive algorithm, we call this technique the *deferred updates*. A typical usage scenario is demonstrated in Algorithm 3.

3.1.4 Worklists and Atomic Operations

There are many cases where it is necessary to create a worklist of items which need to be processed. An example of such a case is the creation of the active sub-mesh in coarsening and swapping, as will be described in Sect. 3.3. Every thread keeps a local list of vertices it has marked as active and all local worklists have to be accumulated into a global worklist, which essentially is the set of all vertices comprising the active sub-mesh.

One approach is to wait for every thread to exit the parallel loop and then perform a prefix sum [4] on the number of vertices in its private list. After that, every thread knows its index in the global worklist at which it has to copy the private list. This method has the disadvantage that every thread must wait for all other threads to

Algorithm 3 Typical example of using the deferred updates mechanism

```
typedef std::vector<Updates> DeferredOperationsList;
int nthreads = omp_get_max_threads();

// Create nthreads collections of deferred operations lists
std::vector< std::vector<DeferredOperationsList> > defOp;
defOp.resize(nthreads);

#pragma omp parallel
{
  // Every OMP thread executes
  int tid = omp_get_thread_num();
  defOp[tid].resize(nthreads);
  // By now, every OMP thread has allocated one list per thread

  // Execute one iteration of an adaptive algorithm in parallel
  // Defer any updates until the end of the iteration
  #pragma omp for
  for(int i=0; i<nVertices; ++i){
    execute kernel(i);
    // Update will be committed by thread i%nthreads
    // where the modulo avoids racing.
    defOp[tid][i%nthreads].push_back(some_update_operation);
  }

  // Traverse all lists which were allocated for thread tid
  // and commit any updates found
  for(int i=0; i<nthreads; ++i){
    commit_all_updates(defOp[i][tid]);
  }
}
```

exit the parallel loop, synchronise with them to perform the prefix sum and finally copy its private data into the global worklist. Profiling data indicates that this way of manipulating worklists is a significant limiting factor towards achieving good scalability.

Experimental evaluation showed that, at least on the Intel Xeon, a better method is based on atomic operations on a global integer variable which stores the size of the worklist needed so far. Every thread which exits the parallel loop increments this integer atomically while caching the old value. This way, the thread knows immediately at which index it must copy its private data and increments the integer by the size of this data, so that the next thread which will access this integer knows in turn its index at which its private data must be copied. Caching the old value before the atomic increment is known in OpenMP terminology as *atomic capture*. Support for atomic capture operations was introduced in OpenMP 3.1. This functionality has also been supported by GNU extensions (intrinsics) since GCC 4.1.2, known under the name *fetch-and-add*. An example of using this technique is shown in Algorithm 4.

Algorithm 4 Example of creating a worklist using OpenMP's atomic capture operations

```
int worklistSize = 0; // Points to end of the global worklist
std::vector<Item> globalWorklist;

// Pre-allocate enough space
globalWorklist.resize(some_appropriate_size);

#pragma omp parallel
{
  std::vector<Item> private_list;

  // Private list - no need to synchronise at end of loop.
  #pragma omp for nowait
  for(all items which need to be processed){
    do_some_work();
    private_list.push_back(item);
  }

  // Private variable - the index in the global worklist
  // at which the thread will copy the data in private_list.
  int idx;

  #pragma omp atomic capture
  {
    idx = worklistSize;
    worklistSize += private_list.size();
  }

  memcpy(&globalWorklist[idx], &private_list[0],
              private_list.size() * sizeof(Item));

}
```

Note the `nowait` clause at the end of the `#pragma omp for` directive. A thread which exits the loop does not have to wait for the other threads to exit. It can proceed directly to the atomic operation. It has been observed that dynamic scheduling for OpenMP for-loops is what works best for most of the adaptive loops in mesh optimisation because of the irregular load balance across the mesh. Depending on the nature of the loop and the chunk size, threads will exit the loop at significantly different times. Instead of having some threads waiting for others to finish the parallel loop, with this approach they do not waste time and proceed to the atomic increment. The profiling data suggests that the overhead or spinlock associated with atomic-capture operations is insignificant.

3.1.5 Reflection on Alternatives

Our initial approach to dealing with structural hazards, data races and propagation of adaptivity was based on a thread-partitioning scheme in which the mesh was split into as many sub-meshes as there were threads available. Each thread was then free to process items inside its own partition without worrying about hazards and races. Items on the halo of each thread-partition were locked (analogous to the MPI parallel strategy); those items would be processed later by a single thread. However, this approach did not result in good scalability for a number of reasons. Partitioning the mesh was a significant serial overhead, which was incurred repeatedly as the adaptive algorithms changed mesh topology and invalidated the existing partitioning. In addition, the single-threaded phase of processing halo items was another hotspot of this thread-partition approach. In line with Amdahl's law, these effects only become more pronounced as the number of threads is increased. For these reasons this thread-level domain decomposition approach was not pursued further.

3.2 Refinement

Every edge can be processed and refined without being affected by what happens to adjacent edges. Being free from structural hazards, the only issue we are concerned with is thread safety when updating mesh data structures. Refining an edge involves the addition of a new vertex to the mesh. This means that new coordinates and metric tensor values have to be appended to `coords` and `metric` and adjacency information in `NNList` has to be updated. The subsequent element split leads to the removal of parent elements from `ENList` and the addition of new ones, which, in turn, means that `NEList` has to be updated as well. Appending new coordinates to `coords`, metric tensors to `metric` and elements to `ENList` is done using the thread worklist strategy described in Sect. 3.1.4, while updates to `NNList` and `NEList` can be handled efficiently using the deferred operations mechanism.

The two stages, namely edge refinement and element refinement, of our threaded implementation are described in Algorithms 5 and 6, respectively. The procedure begins with the traversal of all mesh edges. Edges are accessed using `NNList`, i.e. for each mesh vertex V_i the algorithm visits V_i's neighbours. This means that edge refinement is a directed operation, as edge $\overline{V_i V_j}$ is considered to be different from edge $\overline{V_j V_i}$. Processing the same physical edge twice is avoided by imposing the restriction that we only consider edges for which V_i's ID is less than V_j's ID. If an edge is found to be longer than desired, then it is split in the middle (in metric space) and a new vertex V_n is created. V_n is associated with a pair of coordinates and a metric tensor. It also needs an ID. At this stage, V_n's ID cannot be determined. Once an OpenMP thread exits the edge refinement phase, it can proceed (without synchronisation with the other threads) to fix vertex IDs and append the new data it created to the mesh. The thread captures the number of mesh vertices $index =$

Algorithm 5 Edge-refinement

Global worklist of split edges $\mathscr{W}$, refined_edges_per_element[NElements]
#pragma omp parallel
  private : *split_cnt* $\leftarrow 0$, *newCoords*, *newMetric*, *newVertices*
  #pragma omp for schedule(dynamic)
  for all vertices V_i **do**
    for all vertices V_j adjacent to V_i, $ID(V_i) < ID(V_j)$ **do**
      if *length of edge* $\overline{V_i V_j} > L_{max}$ **then**
        $V_n \leftarrow$ new vertex of split edge $\overline{V_i V_n V_j}$; Append new coordinates, interpolated metric, split edge to *newCoords*, *newMetric*, *newVertices*; *split_cnt* $\leftarrow$ *split_cnt* $+ 1$
  #pragma omp atomic capture
    index $\leftarrow$ *NNodes*; *NNodes* $\leftarrow NNodes +$ *splint_cnt*
  Copy *newCoords* into `coords`, *newMetric* into `metric`
  for all edges $e_i \in$ *newVertices* **do**
    $e_i = \overline{V_i V_n V_j}$; increment ID of V_n by *index*
  Copy *newVertices* into $\mathscr{W}$
  #pragma omp barrier
  #pragma omp parallel for schedule(dynamic)
  for all Edges $e_i \in \mathscr{W}$ **do**
    Replace V_j with V_n in `NNList[`V_i`]`; replace V_i with V_n in `NNList[`V_j`]`
    Add V_i and V_j to `NNList[`V_n`]`
    for all *elements* $E_i \in \{NEList[V_i] \cap NEList[V_j]\}$ **do**
      Mark edge e_i as refined in refined_edges_per_element[E_i].

Algorithm 6 Element refinement phase

#pragma omp parallel
  private : *newElements*
  #pragma omp for schedule(dynamic)
  for all elements E_i **do**
    REFINE_ELEMENT(E_i)
    Append additional elements to *newElements*.
  Resize `ENList`.
  Parallel copy of *newElements* into `ENList`.

NNodes and increments it atomically by the number of new vertices it created. After capturing the index, the thread can assign IDs to the vertices it created and also copy the new coordinates and metric tensors into `coords` and `metric`, respectively.

Before proceeding to element refinement, all split edges are accumulated into a global worklist. For each split edge $\overline{V_i V_j}$, the original vertices V_i and V_j have to be connected to the newly created vertex V_n. Updating `NNList` for these vertices cannot be deferred. Most edges are shared between two elements, so if the update was deferred until the corresponding element were processed, we would run the risk of committing these updates twice, once for each element sharing the edge. Updates can be committed immediately, as there are no race conditions when accessing `NNList` at this point. Besides, for each split edge we find the (usually two) elements

sharing it. For each element, we record that this edge has been split. Doing so makes element refinement much easier, because as soon as we visit an element we will know immediately how many and which of its edges have been split. An array of length `NElements` stores this type of information.

During mesh refinement, elements are visited in parallel and refined independently. It should be noted that all updates to `NNList` and `NEList` are deferred operations. After finishing the loop, each thread uses the worklist method to append the new elements it created to `ENList`. Once again, no thread synchronisation is needed.

Compared to Freitag's task graph approach, this parallel refinement algorithm has the advantage of not requiring any mesh colouring and having low synchronisation overhead as. Additionally, the element refinement phase is based on the results of the edge refinement phase, so we completely avoid having non-conformities and the subsequent need to propagate operations in order to eliminate them.

3.3 Coarsening

Because any decision on whether to collapse an edge is strongly dependent upon what other edges are collapsing in the immediate neighbourhood of elements, an operation task graph for coarsening has to be constructed. Edge collapse is based on the removal of vertices, i.e. the elemental operation for edge collapse is the removal of a vertex. Therefore, the operation task graph $\mathscr{G}$ is the mesh itself.

Figure 5 demonstrates what needs to be taken into account in order to perform parallel coarsening safely. It is clear that adjacent vertices cannot collapse concurrently, so a distance-1 colouring of the mesh is sufficient in order to avoid structural hazards. This colouring also enforces processing of vertices topologically at least *every other one* which prevents skewed elements forming during significant coarsening [8, 17].

An additional consideration is that vertices which are two edges away from each other share some common vertex V_{common}. Removing both vertices at once means that V_{common}'s adjacency list will have to be modified concurrently by two different threads, leading to data races. These races can be avoided using the deferred operations mechanism.

Algorithm 7 illustrates a thread parallel version of mesh edge collapse. Coarsening is divided into two phases: the first sweep through the mesh identifies what edges are to be removed, see Algorithm 8; and the second phase actually applies the coarsening operation, see Algorithm 9. Function *coarsen_identify*(V_i) takes as argument the ID of a vertex V_i, decides whether any of the adjacent edges can collapse and returns the ID of the target vertex V_t onto which V_i should collapse (or a negative value if no adjacent edge can be removed). *coarsen_kernel*(V_i) performs the actual collapse, i.e. removes V_i from the mesh, updates vertex adjacency information and removes the two deleted elements from the element list.

Algorithm 7 Edge collapse

```
Allocate dynamic_vertex, worklist.
#pragma omp parallel
  #pragma omp for schedule(static)
  for all vertices V_i do dynamic_vertex[V_i] ← −2
  Colour mesh
  repeat
    #pragma omp for schedule(dynamic)
    for all vertices V_i do
      if dynamic_vertex[V_i] == −2 then
        dynamic_vertex[V_i] ←COARSEN_IDENTIFY(V_i)
    if dynamic vertex count == 0 then break
    I_m ← maximal independent set of dynamic vertices
    #pragma omp for schedule(dynamic)
    for all V_i ∈ I_m do
                                  ▷ mark all neighbours for re-evaluation
      for all vertices V_j ∈ NNList[V_i] do
        dynamic_vertex[V_j] ← −2
      dynamic_vertex[V_i] ← −1
      COARSEN_KERNEL(V_i)
    Commit deferred operations.
    Repair colouring
  until true
```

Algorithm 8 coarsen_identify

```
procedure COARSEN_IDENTIFY(V_i)
  S_i ← the set of all edges connected to V_i
  S^0 ← S_i
  repeat
    E_j ← shortest edge in S^j
    if length of E_j > L_min then          ▷ if shortest edge is of acceptable
      return -1                            ▷ length, no edge can be removed
    V_t ← the other vertex that bounds E_j
    evaluate collapse of E_j with the collapse of V_i onto V_t
    if (∀ edges ∈ S_i ≤ L_max) and (∄ inverted elements) then
      return V_t
    else
      remove E_j from S^j                  ▷ E_j is not a candidate for collapse
  until S_i = ∅
```

Algorithm 9 Coarsen_kernel with deferred operations

procedure COARSEN_KERNEL(V_i)
  $V_t \leftarrow$ *dynamic_vertex*$[V_i]$
  removed_elements $\leftarrow$ `NEList[`V_i`]` $\cap$ `NEList[`V_t`]`
  common_patch $\leftarrow$ `NNList[`V_i`]` $\cap$ `NNList[`V_t`]`
  for all $E_i \in removed\_elements$ **do**
    $V_o \leftarrow$ the other vertex of $E_i = \widehat{V_i V_t V_o}$
    `NEList[`V_o`].erase(`E_i`)` ▷ deferred operation
    `NEList[`V_t`].erase(`E_i`)` ▷ deferred operation
    `NEList[`V_i`].erase(`E_i`)`
    `ENList[3*`E_i`]` $\leftarrow -1$ ▷ erase element by resetting its first vertex
  for all $E_i \in$ NEList$[V_i]$ **do**
    replace V_i with V_t in `ENList[3*`E_i`+{0,1,2}]`
    `NEList[`V_t`].add(`E_i`)` ▷ deferred operation
  remove V_i from `NNList[`V_t`]` ▷ deferred operation
  for all $V_c \in common\_patch$ **do**
    remove V_i from `NNList[`V_c`]` ▷ deferred operation
  for all $V_n \notin common\_patch$ **do**
    replace V_i with V_t in `NNList[`V_n`]`
    add V_n to `NNList[`V_t`]` ▷ deferred operation
  `NNList[`V_i`].clear()`
  `NEList[`V_i`].clear()`

Parallel coarsening begins with the initialisation of array *dynamic_vertex* which is defined as:

$$dynamic\_vertex[V_i] = \begin{cases} -1 & V_i \text{ cannot collapsed,} \\ -2 & V_i \text{ must be re-evaluated,} \\ V_t & V_i \text{ is about to collapse onto } V_t. \end{cases}$$

At the beginning, the whole array is initialised to -2, so that all mesh vertices will be considered for collapse.

In each iteration of the outer coarsening loop, *coarsen_identify_kernel* is called for all vertices which have been marked for (re-)evaluation. Every vertex for which $dynamic\_vertex[V_i] \geq 0$ is said to be *dynamic* or *active*. At this point, a reduction in the total number of active vertices is necessary to determine whether there is anything left for coarsening or the algorithm should exit the loop.

Next up, we find the maximal independent set of active vertices $\mathscr{I}_m$. Working with independent sets not only ensures safe parallel execution, but also enforces the *every other vertex* rule. For every active vertex $V_r \in \mathscr{I}_m$ which is about to collapse, the local neighbourhood of all vertices V_a formerly adjacent to V_r changed and target vertices $dynamic\_vertex[V_a]$ may not be suitable choices any more. Therefore, when V_r is erased, all its neighbours are marked for re-evaluation. This is how propagation of coarsening is implemented.

Algorithm 9 describes how the actual coarsening takes place in terms of modifications to mesh data structures. Updates which can lead to race conditions

have been pointed out. These updates are deferred until the end of processing of the independent set. Before moving to the next iteration, all deferred operations are committed and colouring is repaired because edge collapse may have introduced inconsistencies.

3.4 Swapping

The data dependencies in edge swapping are virtually identical to those of edge coarsening. Therefore, it is possible to reuse the same thread parallel algorithm as for coarsening in the previous section with slight modifications

In order to avoid maintaining edge-related data structures (e.g. edge-node list, edge-edge adjacency lists etc.), an edge can be expressed in terms of a pair of vertices. Just like in refinement, we define an edge E_{ij} as a pair of vertices (V_i, V_j), with $ID(V_i) < ID(V_j)$. We say that E_{ij} is outbound from V_i and inbound to V_j. Consequently, the edge E_{ij} can be marked for swapping by adding V_j to $marked\_edges[V_i]$. Obviously, a vertex V_i can have more than one outbound edge, so unlike *dynamic_vertex* in coarsening, *marked_edges* in swapping needs to be a vector of sets.

The algorithm begins by marking all edges. It then enters a loop which is terminated when no marked edges remain. The maximal independent set $\mathscr{I}_m$ of active vertices is calculated. A vertex is considered active if at least one of its outbound edges is marked. Following that, threads process all active vertices of $\mathscr{I}_m$ in parallel. The thread processing vertex V_i visits all edges in $marked\_edges[V_i]$ one after the other and examines whether they can be swapped, i.e. whether the operation will improve the quality of the two elements sharing that edge. It is easy to see that swapping two edges in parallel which are outbound from two independent vertices involves no structural hazards.

Propagation of swapping is similar to that of coarsening. Consider the local patch in Fig. 3 and assume that a thread is processing vertex V_0. If edge $\overline{V_0V_1}$ is flipped, the two elements sharing that edge change in shape and quality, so all four edges surrounding those elements (forming the rhombus in bold) have to be marked for processing. This is how propagation is implemented in swapping.

One last difference between swapping and coarsening is that $\mathscr{I}_m$ needs to be traversed more than once before proceeding to the next one. In the same example as above, assume that all edges adjacent to V_0 are outbound and marked. If edge $\overline{V_0V_1}$ is flipped, adjacency information for V_1, V_2 and V_3 has to be updated. These updates have to be deferred because another thread might try to update the same lists at the same time (e.g. the thread processing edge $\overline{V_CV_1}$). However, not committing the changes immediately means that the thread processing V_0 has a stale view of the local patch. More precisely, `NEList[`V_2`]` and `NEList[`V_3`]` are invalid and cannot be used to find what elements edges $\overline{V_0V_2}$ and $\overline{V_0V_3}$ are part of. Therefore, these two edges cannot be processed until the deferred operations have been committed. On the other hand, the rest of V_0's outbound edges are free to be processed. Once all

Algorithm 10 Thread-parallel mesh smoothing

```
repeat
  relocate_count ← 0
  for colour = 1 → k do
    #pragma omp for schedule(static)
    for all i ∈ V^c do
                                          ▷ move_success is true if vertex was relocated,
      move_success ← smooth_kernel(i)     ▷ false otherwise.
      if move_success then
        relocate_count ← relocate_count + 1
until (n ≥ max_iteration)or(relocate_count = 0)
```

threads have processed whichever edges they can for all vertices of the independent set, deferred operations are committed and threads traverse the independent set again (up to two more times in 2D) to process what had been skipped before.

3.5 *Smoothing*

Algorithm 10 illustrates the colouring based algorithm for mesh smoothing and is described in greater detail in [16]. In this algorithm the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ consists of sets of vertices $\mathcal{V}$ and edges $\mathcal{E}$ that are defined by the vertices and edges of the computational mesh. By computing a vertex colouring of $\mathcal{G}$ we can define independent sets of vertices, $\mathcal{V}^c$, where c is a computed colour. Thus, all vertices in $\mathcal{V}^c$, for any c, can be updated concurrently without any race conditions on dependent data. This is clear from the definition of the smoothing kernel in Sect. 2.4.4. Hence, within a node, thread-safety is ensured by assigning a different independent set $\mathcal{V}^c$ to each thread.

4 Results

In order to evaluate the parallel performance, an isotropic mesh was generated on the unit square with using approximately 200×200 vertices. A synthetic solution ψ is defined to vary in time and space:

$$\psi(x, y, t) = 0.1 \sin(50x + 2\pi t/T) + \arctan(-0.1/(2x - \sin(5y + 2\pi t/T))), \tag{5}$$

where T is the period. An example of the field at $t = 0$ is shown in Fig. 6. This is a good choice as a benchmark as it contains multi-scale features and a shock front. These are the typical solution characteristics where anisotropic adaptive mesh methods excel.

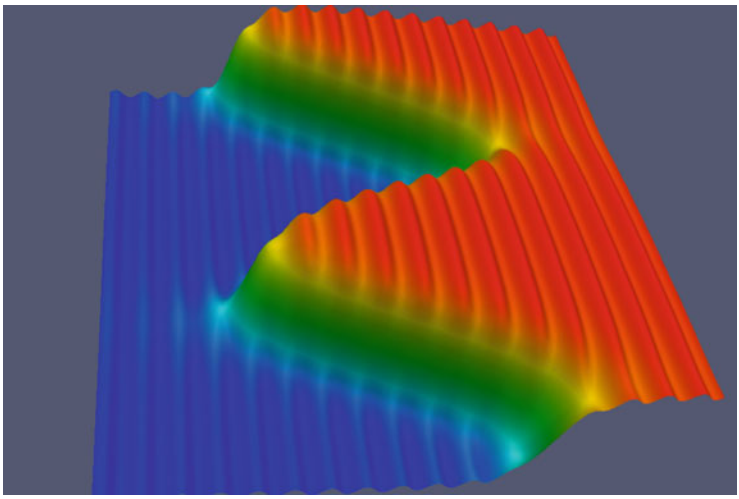

Fig. 6 Benchmark solution field

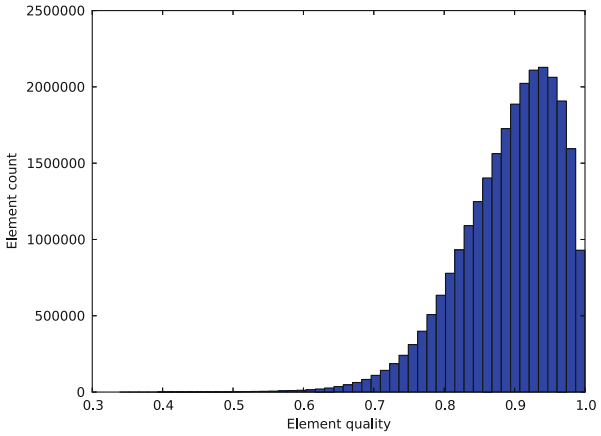

Fig. 7 Histogram of element qualities aggregated over all iterations

Because mesh adaptation has a very irregular workload we simulate a time varying scenario where t varies from 0 to 51 in increments of unity and we use the mean and standard deviations when reporting performance results. To calculate the metric we used the L^p-norm as described by [6], where $p \equiv 2$. The number of mesh vertices and elements maintains an average of approximately $250k$ and $500k$ respectively. As the field evolves all of the adaptive operations are heavily used, thereby giving an overall profile of the execution time.

In order to demonstrate the correctness of the adaptive algorithm we plot a histogram (Fig. 7) showing the quality of all element aggregated over all time steps.

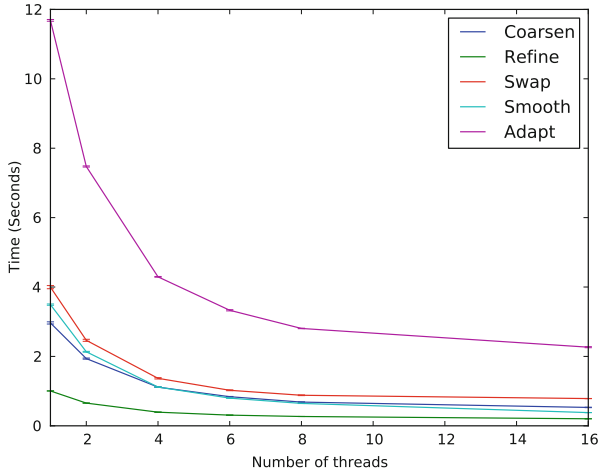

Fig. 8 Wall time for each mesh adaptation phase

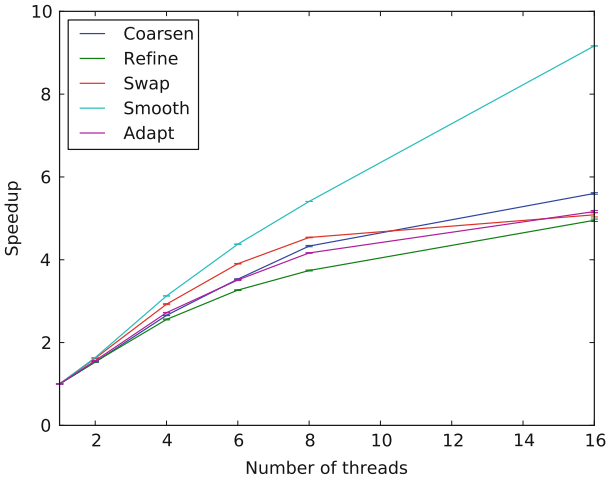

Fig. 9 Speedup for each mesh adaptation phase

We can see that the vast majority of the elements are of very quality. The lowest quality element had a quality of 0.34, and in total only ten elements out of 26 million have a quality less than 0.4.

The benchmarks were run on a Intel(R) Xeon(R) E5-2650 CPU. The code was compiled using the Intel compiler suite, version 14.0.1 and with the compiler flags `-Ofast`. In all cases we used Intel's thread-core affinity support - specifically `scatter` which distributes the threads as evenly as possible across the entire system.

Figures 8, 9 and 10 show the wall time, speedup and efficiency of each phase of mesh adaptation. Simulations using between 1 and 8 cores are run on a single

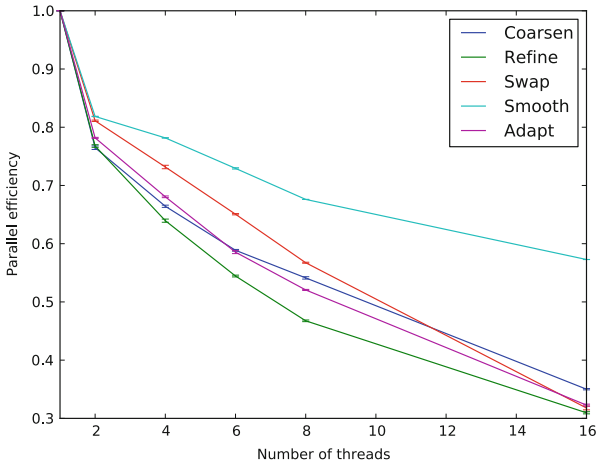

Fig. 10 Parallel efficiency for each mesh adaptation phase

socket while the 16 core simulation runs across two CPU sockets and thereby incurring NUMA overheads. From the results we can see that all operations achieve good scaling, including for the 16 core NUMA case. The dominant factors limiting scaling are the number of synchronisations and load-imbalances. Even in the case of mesh smoothing, which involves the least data-writes, the relatively expensive optimisation kernel is only executed for patches of elements whose quality falls below a minimum quality tolerance. Indeed, the fact that mesh refinement, coarsening and refinement are comparable is very encouraging as it indicates that despite the invasive nature of the operations on these relatively complex data structures it is possible to get good intra-node scaling.

5 Conclusions

This paper is the first to examine the scalability of anisotropic mesh adaptivity using a thread-parallel programming model and to explore new parallel algorithmic approaches to support this model. Despite the complex data dependencies and inherent load imbalances we have shown it is possible to achieve practical levels of scaling. To achieve this two key ingredients were required. The first was to use colouring to identify maximal independent sets of tasks that would be performed concurrently. In principle this facilitates scaling up to the point that the number of elements of the independent set is equal to the number of available threads. The second important factor contributing to the scalability was the use of worklists and deferred whereby updates to the mesh are added to worklists and applied in parallel at a later phase of an adaptive sweep. This avoids the majority of serial overheads otherwise incurred with updating mesh data structures.

While the algorithms presented are for 2D anisotropic mesh adaptivity, we believe many of the algorithmic details carry over to the 3D case as the challenges associated with exposing a sufficient degree of parallelism are very similar.

Acknowledgements The authors would like to thank Fujitsu Laboratories of Europe Ltd. and EPSRC grants EP/I00677X/1 and EP/L000407/1 for supporting this work.

References

1. Agouzal, A., Lipnikov, K., Vassilevski, Y.: Adaptive generation of quasi-optimal tetrahedral meshes. East West J. Numer. Math. **7**(4), 223–244 (1999)
2. Alauzet, F.: Size gradation control of anisotropic meshes. Finite Elem. Anal. Des. **46**(1), 181–202 (2010)
3. Alauzet, F., Li, X., Seol, E.S., Shephard, M.S.: Parallel anisotropic 3D mesh adaptation by mesh modification. Eng. Comput. **21**(3), 247–258 (2006)
4. Blelloch, G.E.: Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (1990)
5. Buscaglia, G.C., Dari, E.A.: Anisotropic mesh optimization and its application in adaptivity. Int. J. Numer. Methods Eng. **40**(22), 4119–4136 (1997)
6. Chen, L., Sun, P., Xu, J.: Optimal anisotropic meshes for minimizing interpolation errors in L^p-norm. Math. Comput. **76**(257), 179–204 (2007)
7. Coupez, T., Digonnet, H., Ducloux, R.: Parallel meshing and remeshing. Appl. Math. Model. **25**(2), 153–175 (2000)
8. De Cougny, H., Shephard, M.S.: Parallel refinement and coarsening of tetrahedral meshes. Int. J. Numer. Methods Eng. **46**(7), 1101–1125 (1999)
9. Dongarra, J.: What you can expect from exascale computing. In: International Supercomputing Conference (ISC'11), Hamburg (2011)
10. Field, D.A.: Laplacian smoothing and Delaunay triangulations. Commun. Appl. Numer. Methods **4**, 709—712 (1988)
11. Freitag, L., Ollivier-Gooch, C.: A comparison of tetrahedral mesh improvement techniques. In: Fifth International Meshing Roundtable (1996)
12. Freitag, L., Jones, M., Plassmann, P.: An efficient parallel algorithm for mesh smoothing. In: Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories, Citeseer, pp. 47–58 (1995)
13. Freitag, L.F., Jones, M.T., Plassmann, P.E.: The scalability of mesh improvement algorithms. In: Improvement Algorithms. IMA Volumes in Mathematics and Its Applications, pp. 185–212. Springer, New York (1998)
14. Frey, P.J., Alauzet, F.: Anisotropic mesh adaptation for cfd computations. Comput. Methods Appl. Mech. Eng. **194**(48), 5068–5082 (2005)
15. Gebremedhin, A.H., Manne, F.: Scalable parallel graph coloring algorithms. Concurrency Pract. Experience **12**(12),1131–1146 (2000)
16. Gorman, G., Southern, J., Farrell, P., Piggott, M., Rokos, G., Kelly, P.: Hybrid OpenMP/MPI anisotropic mesh smoothing. In: Proceedings of the International Conference on Computational Science. Procedia Computer Science, vol. 9, pp. 1513–1522 (2012)
17. Li, X., Shephard, M., Beall, M.: 3D anisotropic mesh adaptation by mesh modification. Comput. Methods Appl. Mech. Eng. **194**(48-49), 4915–4950 (2005)
18. Pain, C.C., Umpleby, A.P., de Oliveira, C.R.E., Goddard, A.J.H.: Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. Comput. Methods Appl. Mech. Eng. **190**(29-30), 3771–3796 (2001)

19. Rabenseifner, R., Hager G, Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2009, pp. 427–436. IEEE (2009)
20. Rokos, G., Gorman, G., Kelly, P.H.J.: Accelerating anisotropic mesh adaptivity on nVIDIA's CUDA using texture interpolation. In: Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, Euro-Par'11, pp. 387–398. Springer, Berlin (2011). http://dl.acm.org/citation.cfm?id=2033408.2033453
21. Tam, A., Ait-Ali-Yahia, D., Robichaud, M., Moore, M., Kozel, V., Habashi, W.: Anisotropic mesh adaptation for 3D flows on structured and unstructured grids. Comput. Methods Appl. Mech. Eng. **189**(4), 1205–1230 (2000)
22. Vasilevskii, Y., Lipnikov, K.: An adaptive algorithm for quasioptimal mesh generation. Comput. Math. Math. Phys. **39**(9), 1468–1486 (1999)