Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1513 – 1522

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Hybrid OpenMP/MPI anisotropic mesh smoothing

G.J. Gorman[a,*], J. Southern[b], P.E. Farrell[a], M.D. Piggott[a], G. Rokos[a], P.H.J. Kelly[a]

[a]*Applied Modelling and Computation Group, Department of Earth Science and Engineering, Imperial College London, SW7 2AZ, UK*
[b]*Fujitsu Laboratories of Europe, Hayes Park Central, Hayes End Road, Hayes, Middlesex, UB4 8FE, UK*

Abstract

Mesh smoothing is an important algorithm for the improvement of element quality in unstructured mesh finite element methods. A new optimisation based mesh smoothing algorithm is presented for anisotropic mesh adaptivity. It is shown that this smoothing kernel is very effective at raising the minimum local quality of the mesh. A number of strategies are employed to reduce the algorithm's cost while maintaining its effectiveness in improving overall mesh quality. The method is parallelised using hybrid OpenMP/MPI programming methods, and graph colouring to identify independent sets. Different approaches are explored to achieve good scaling performance within a shared memory compute node.

Keywords: unstructured mesh, mesh smoothing, ccNUMA, parallel, OpenMP, MPI

1. Introduction

Computational mesh resolution is often the limiting factor in simulation accuracy. Indeed, being able to accurately resolve physical processes at the small scale, coupled with larger scale dynamics, is key to improving the fidelity of numerical models across a wide range of applications, from earth system components used in climate prediction to the simulation of cardiac electrophysiology [1, 2]. Since many of these applications include a strong requirement to conform to complex geometries or to resolve a multi-scale solution, the numerical methods used to model them often favour the use of unstructured meshes and finite element or finite volume discretisation methods over structured grid alternatives. However, this flexibility introduces complications of its own, such as the management of mesh quality and additional computational overheads arising from indirect addressing.

Mesh adaptivity methods provide an important means to control solution error by focusing mesh resolution in regions of the computational domain where and when it is required. In ideal circumstances, where there is a method to estimate the solution error, mesh adaptivity allows one to compute a solution to a specified error tolerance while using the minimum resolution everywhere in the domain. However, for many practical applications the available computational resources are bounded, and so instead these algorithms are used to compute the most accurate solution possible for a given resolution.

Parallel computing — in order to make use of larger compute resources — provides an obvious source of further improvements in accuracy. However, this comes at the cost of further overheads, including the need to manage

*Corresponding author
Email address: g.gorman@imperial.ac.uk (G.J. Gorman)

doi:10.1016/j.procs.2012.04.166

the distribution of the mesh over the available compute resources and the synchronisation of halo regions. Over the past ten years there has been a trend towards an increasing number of cores per node in the world's most powerful supercomputers— and it is assumed that the nodes of a future exascale supercomputer will each contain thousands or even tens of thousands of cores [3]. On such architectures, a hybrid programming model that uses thread-based parallelism to exploit shared memory within a node and a message passing paradigm for distributed memory between nodes can often be the superior solution over pure message passing approaches. This can be due to reduced communication needs, memory consumption, improved load balance or other algorithmic changes [4].

A crucial component of many unstructured mesh adaptivity algorithms is mesh smoothing. This provides a powerful, if heuristic, approach to improve mesh quality. A diverse range of approaches to mesh smoothing have been proposed [5, 6, 7, 8, 9, 10, 11, 12, 13]. Effective algorithms for parallelising mesh smoothing extracting concurrency have also been proposed within the context of a Parallel Random Access Machine (PRAM) computational model [14]. Mesh smoothing is frequently used in isolation as well as with other adaptive mesh operations. Thus, the development of a hybrid mesh smoothing algorithm represents a natural first step on the path to a complete hybrid library for mesh adaptivity.

The novel contributions of this paper are to introduce a new optimisation based mesh smoothing algorithm for anisotropic mesh adaptivity and to highlight implementation and algorithmic approaches to achieving good scaling at the threaded level. The smoothing kernel solves a non-linear optimisation problem by differentiating the local mesh quality with respect to mesh node position and employing hill climbing to maximise the quality of the worst local element. It is shown that this approach is effective at raising globally the minimum element quality of the mesh. The method is parallelised for hybrid OpenMP/MPI using standard colouring techniques. Two different implementation and algorithmic strategies are presented which can significantly boost the performance and scaling of the methods.

The paper is laid out at follows. Section 2 introduces a number of vertex smoothing kernels, before describing the optimisation based smoothing kernel. Section 3 describes the parallel algorithm that applies these smoothing kernels. A number of important implementation details are highlighted which can significantly impact performance depending on the details of the parallel algorithm. In section 4 some numerical experiments are presented which demonstrate the method and highlight some of the issues discussed, and we finish with some conclusions in Section 5. All the software used in this research is freely available under an open source license and can be downloaded from Launchpad[1].

2. Vertex smoothing kernel

2.1. 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 Figure 1). Probably the best known heuristic for mesh smoothing is Laplacian smoothing, first proposed by Field [5]. 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 length and angle are performed with respect to a metric tensor field that describes the desired size and orientation of mesh elements (e.g. [13]). Therefore, in general, the proposed new position of a vertex $\vec{v}_i^{\mathcal{L}}$ is given by

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

where $\vec{v}_j$, $j = 1, \ldots, J$, are the vertices connected to $\vec{v}_i$ 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 [5], the application of pure Laplacian smoothing can actually decrease useful local element quality metrics; 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 et al. [8] (while Freitag et al. only discuss this for Euclidean geometry it is straightforward to extend to the

[1] https://launchpad.net/pragmatic

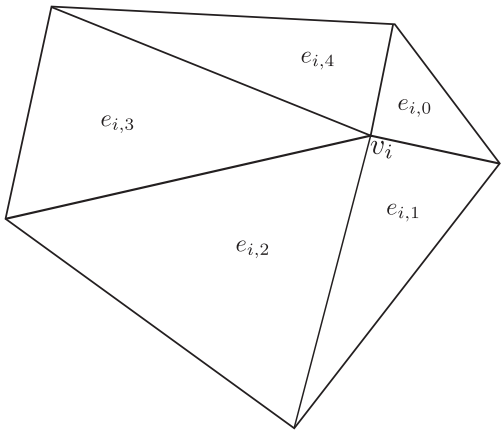


Figure 1: Local mesh patch: $\vec{v}_i$ is the vertex being relocated; $\{e_{i,0}, \ldots, e_{i,m}\}$ is the set of elements connected to $\vec{v}_i$.

more general case), is summarised in Algorithm 1. 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):

$$Q(\vec{v}_i) \equiv \|\boldsymbol{q}\|_\infty \tag{2}$$

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

Many measures of element quality have been proposed. In general, for mesh generation applications Euclidean geometric metrics are considered [15, 16]. However, these metrics do not take into account characteristics of the solution. Therefore, other measures of element quality have been proposed which do take into consideration both the shape and size of the elements required for controlling solution errors.

In the work described here, we use the element quality measure for 2D simplexes proposed by Vasilevskii et al. [17]:

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

where $|\triangle|_M$ is the area of element $\triangle$ and $|\partial\triangle|_M$ is its perimeter as measured with respect to the non-Euclidean metric 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, $|\triangle| = l^2\sqrt{3}/4$ 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:

$$F(x) = (\min(x, 1/x)(2 - \min(x, 1/x)))^3\,, \tag{4}$$

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, e.g. an equilateral triangle with sides of unit length, and $II < 1$ otherwise. Hence, taken together, the two factors making up Q yield a maximum value of unity for an equilateral triangle with edges of unit length, and Q is less than one otherwise.

2.2. Optimisation based smoothing

A much more effective (albeit more computationally expensive) method of increasing the local element quality is to solve a local non-smooth optimisation problem, as shown in Algorithm 2. For this it is assumed that the derivatives of non-inverted element quality are smooth, although the patch quality given in equation (2) is not. Note that while $q^M(\triangle)$ as defined in equation (3) is not differentiable everywhere, it is differentiable almost everywhere (as F is not differentiable at x=1). The algorithm proceeds by stepping in the direction of the quality gradient of the worst element,

Algorithm 1 Smart smoothing kernel: a Laplacian smooth operation is accepted only if it does not reduce the infinity norm of local element quality.

$\vec{v}_i^{\,0} \leftarrow \vec{v}_i$
$quality^0 \leftarrow Q(\vec{v}_i)$
$n \leftarrow 1$ {Initialise iteration counter}
$\vec{v}_i^{\,n} \leftarrow \vec{v}_i^{\,\mathcal{L}}$ {Initialise new vertex location using a Laplacian smooth.}
$M_i^n \leftarrow metric\_interpolation(\vec{v}_i^{\,n})$ {Interpolate metric tensor at proposed location.}
$quality^n = Q(\vec{v}_i^{\,n})$ {Calculate the new local quality for this relocation.}
{Terminate loop if maximum number of iterations is reached or an improvement to the mesh is made.}
while $(n \leq max\_iteration)$**and**$(quality_i^n - quality_i^0 < \sigma_q)$ **do**
  $\vec{v}_i^{\,n+1} \leftarrow (\vec{v}_i^{\,n} + \vec{v}_i^{\,0})/2$
  $M_i^{n+1} \leftarrow metric\_interpolation(\vec{v}_i^{\,n+1})$
  $quality^{n+1} \leftarrow Q(\vec{v}_i^{\,n+1})$
  $n = n + 1$
{Update vertex location and metric tensor for that vertex if mesh is improved.}
if $quality_i^n - quality_i^0 > \sigma_q$ **then**
  $\vec{v}_i \leftarrow \vec{v}_i^{\,n}$
  $M_i \leftarrow M_i^n$

$\vec{s}$. The step size, α, is determined by first using a first order Taylor expansion to model how the quality of the worst element q' will vary along the search direction:

$$q' = q + \alpha \vec{\nabla} q \cdot \vec{s}. \tag{5}$$

With the choice of $\vec{s} \equiv \vec{\nabla} q / |\vec{\nabla} q|$, this becomes

$$q' = q + \alpha |\vec{\nabla} q|. \tag{6}$$

Similarly, the qualities of the other elements q'_e can be modeled with a Taylor expansion, where we consider the elements quality gradient projected onto the search direction:

$$q'_e = q_e + \alpha \vec{s} \cdot \vec{\nabla} q_e. \tag{7}$$

When the quality function of the worst element intersects with the quality function of another element (i.e. when $q' = q'_e$ for some e), we have a point beyond which improving the quality of the worst element would degrade the quality of the patch as a whole. Therefore, we equate the two expressions and solve for α:

$$\alpha = \frac{q - q_e}{\vec{s} \cdot \vec{\nabla} q_e - |\vec{\nabla} q|}. \tag{8}$$

Subsequently, a new search direction is chosen and another step is taken. This is continued until the algorithm converges. The convergence criterion chosen is either a limit on the maximum number of iterations or when the projected improvement in quality falls below some tolerance σ_q.

3. Parallel algorithm

In view of the switch to multi-core nodes, mesh smoothing methods based on traditional task-based parallelism (often using MPI) require an update in order to be able to fully exploit the increased level of intra-node parallelism offered by the latest generation of supercomputers. Purely thread-based parallelism (using OpenMP or pthreads) can exploit the shared memory within a node but cannot be scaled beyond a single node. So, in this work a hybrid OpenMP/MPI algorithm for mesh scaling is proposed in order to optimally exploit both intra- and inter-node parallelism. OpenMP is preferred over, e.g., pthreads due to its greater potential for use with co-processors such as Intel MIC [18] and its simpler interface (via pragmas in `C` code), that simplifies code maintenance. The hybrid model is

Algorithm 2 Optimisation based smoothing kernel: local element quality is improved by solving a local non-smooth optimisation problem.

smart_smooth_kernel($\vec{v}_i$, M_i) {Initialise by applying smart Laplacian smooth to try to improve start position.}
$quality^0 \leftarrow Q(\vec{v}_i)$
$n \leftarrow 0$
repeat
  $\{\vec{\nabla} q_{e_0}, ..., \vec{\nabla} q_{e_j}, ...\}$ {Calculate initial element quality gradients.}
  $\vec{s}^n = \nabla \widehat{\vec{q}_{e_j|q_{e_j} \equiv Q(\vec{v}_i)}}$ {Choose search direction to be that of the quality gradient of the worst local element.}
  $\alpha = nearest\_discontinuity()$ {Calculate α using equation (8).}
  $\vec{v}_i^{n+1} = \vec{v}_i^n + \alpha \vec{s}^n$ {Propose new location for vertex.}
  $M_i^{n+1} \leftarrow metric\_interpolation(\vec{v}_i^{n+1})$ {Interpolate metric tensor at this new location.}
  $quality^{n+1} \leftarrow Q(\vec{v}_i^{n+1})$ {Evaluate local quality using proposed location.}
  {If the improvement is greater than σ_q then accept proposed location and update metric tensor.}
  if $quality_i^{n+1} - quality_i^n > \sigma_q$ **then**
    $\vec{v}_i \leftarrow \vec{v}_i^n$
    $M_i \leftarrow M_i^n$
  $n = n + 1$
  {Continue hill climbing until maximum number of iterations or until there are no further improvement to local mesh quality.}
until $(n \geq max\_iteration)$**or**$(quality_i^n - quality_i^{n-1} < \sigma_q)$

Algorithm 3 Hybrid parallel loop

repeat
  $relocate\_count \leftarrow 0$
  for $colour = 1 \rightarrow k$ **do**
    #pragma omp for schedule(static)
    for all $i \in \mathcal{V}^c$ **do**
      $move\_success \leftarrow smooth\_kernel(i)$ {$move\_success$ is `true` if vertex was relocated, `false` otherwise.}
      if $move\_success$ **then**
        $relocate\_count \leftarrow relocate\_count + 1$
    $update\_halo()$ {MPI: Update state of domain decomposition halo vertices.}
until $(n \geq max\_iteration)$**or**$(relocate\_count = 0)$

preferred over partitioned global address space (PGAS) alternatives such as Co-Array Fortran or Unified Parallel C (UPC) because of the lower overhead of adding OpenMP parallelism to an MPI code (compared to re-writing an entire application in a PGAS language) and, further, due to the more limited PGAS support available in existing toolsets (e.g. profilers and debuggers).

3.1. Concurrency through colouring

Algorithm 3 shows a basic hybrid parallel algorithm used for mesh smoothing. 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 Section 2. Hence, within a node, thread-safety is ensured by assigning a different independent set $\mathcal{V}^c$ to each thread.

Since domain decomposition methods are generally used when the finite element (or finite volume) methods are parallelised for distributed memory parallel computers using MPI, it is also important that the colouring is consistent between subdomains so that we can avoid either threads or processes updating interdependent data within the same

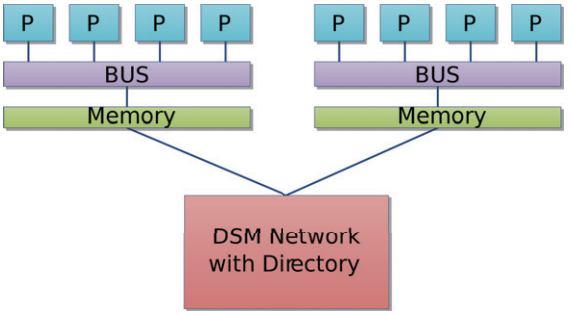


Figure 2: One possible architecture of a two socket NUMA system. Each **P** is a separate core with its own cache; each is connected via a shared memory bus to a local memory node; the operating system gives a continuous memory address space via the virtual memory manager. Processors are connected to different locations in memory through connections of varying latency, so different CPU's have different memory access times based on their location. Image courtesy of Wikimedia Commons.

iteration. In order to ensure that this was the case we used a parallel graph colouring algorithm developed by Boman et al. [19] and implemented in Zoltan [20] to calculate a k-colouring of $\mathcal{G}$.

3.2. Enhancing data locality

Multi-core machines are typically Non-Uniform Memory Architectures (NUMA). This means that memory latency (access time) depends on the physical memory location relative to a processor. Within a NUMA system, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors, see Figure 2. Therefore, it is important that memory be allocated from specific memory nodes, so that software running on the machine may take advantage of NUMA locality. To achieve this, the Linux kernel's memory is partitioned by memory node. By default, page faults are satisfied by memory attached to the page-faulting CPU. Because the first CPU to touch the page will be the CPU that faults the page in, this default policy is called *first touch*.

At the level of application code the required page layout can be achieved with minimum impact on the software. The following snippet of `C` code illustrates how this is done:

```
array = malloc(array_size);
#pragma omp for schedule(static)
for(i=0;i<array_length;i++)
  array[i] = 0;
```

There are edge effects which impact the quality of the data locality with this approach. When `malloc` is called it may return a pointer to a page of memory that has already been bound to a memory node, in which case the page will remain bound to the same memory node regardless of which CPU subsequently writes to it. In addition, there will in general be a page that strides data scheduled for threads running on different sockets and therefore memory nodes. This could be fixed by more complex data structures and access methods; but this would be at the cost of much more complex code.

The native thread queue scheduling algorithm is not optimal for high performance computing. Processor affinity (introduced in Linux kernel 2.5.8) is a modification of the native kernel scheduling algorithm which allows users to prescribe at run time a hard affinity between threads and CPU's. Each thread in the queue has a tag indicating its preferred CPU (or core). Processor affinity takes advantage of the fact that some remnants of a process may remain in one processor's state (in particular, memory pages and cache) from the last time the thread ran, and so scheduling it to run on the same processor the next time could result in the process running more efficiently than if it were to run on another processor. Overall system efficiency increases by reducing performance-degrading situations such as fetching memory from memory nodes which are not directly connected to the CPU and cache misses.

Algorithm 4 Progressive hybrid parallel loop

```
relocate_count ← 0
active_vertices[number_vertices] ← true
for colour = 1 → k do
  #pragma omp for schedule(static)
  for all i ∈ V^c do
    move_success ← smooth_kernel(i) {move_success is true if vertex was relocated, false otherwise.}
    active_vertices[i] ← false
    if move_success then
      relocate_count ← relocate_count + 1
      for all vertices j connected to i do
        active_vertices[j] ← true
  update_halo()
repeat
  relocate_count ← 0
  for colour = 1 → k do
    #pragma omp for schedule(dynamic)
    for all i ∈ V^c do
      if active_vertices[i] then
        move_success ← smooth_kernel(i)
        active_vertices[i] ← false
        if move_success then
          relocate_count ← relocate_count + 1
          for all vertices j connected to i do
            active_vertices[j] ← true
    update_halo()
until (n ≥ max_iteration)or(relocate_count = 0)
```

Mesh vertices are renumbered to improve cache coherency for loops over vertices using a fill-reducing ordering method (specifically `METIS_NodeND`) from [21]. Elements are subsequently renumbered in order of increasing node number. This also complements the memory placement methods described above. However, it would be interesting in the future to explore how different strategies might be best employed to account for the fact that iterating through independent sets might not be good for cache coherency.

3.3. Progressive domain masking

One approach to reduce the overall computational cost of mesh smoothing is to use a mask to identify vertices which do not need to be updated. One should observe that after a vertex has been relocated, then it does not need to be re-evaluated unless one of its adjacent vertices has been subsequently relocated. The revised algorithm is summarised in Algorithm 4. Note that within the second loop block over colours, the OpenMP `for` schedule has been set to dynamic, although it was set to static previously in order to maximise data locality. The reason for this is that as the active set decreases a load imbalance would emerge in a static schedule. The dynamic schedule behaves more like a task farm, so the loss in data locality is compensated for through improved load balancing of the threads.

4. Numerical experiments

All the numerical experiments were conducted on a dual-socket Intel Westmere server, with each socket consisting of a 6-core Xeon CPU X5650 @ 2.67GHz. The code was compiled with the Intel compiler suite version 11.1 and with the compile options `-O2 -openmp`. Each benchmark was executed 10 times on exclusive access compute nodes, performance values are calculated as the mean and error bars indicate the standard error of the measurements.

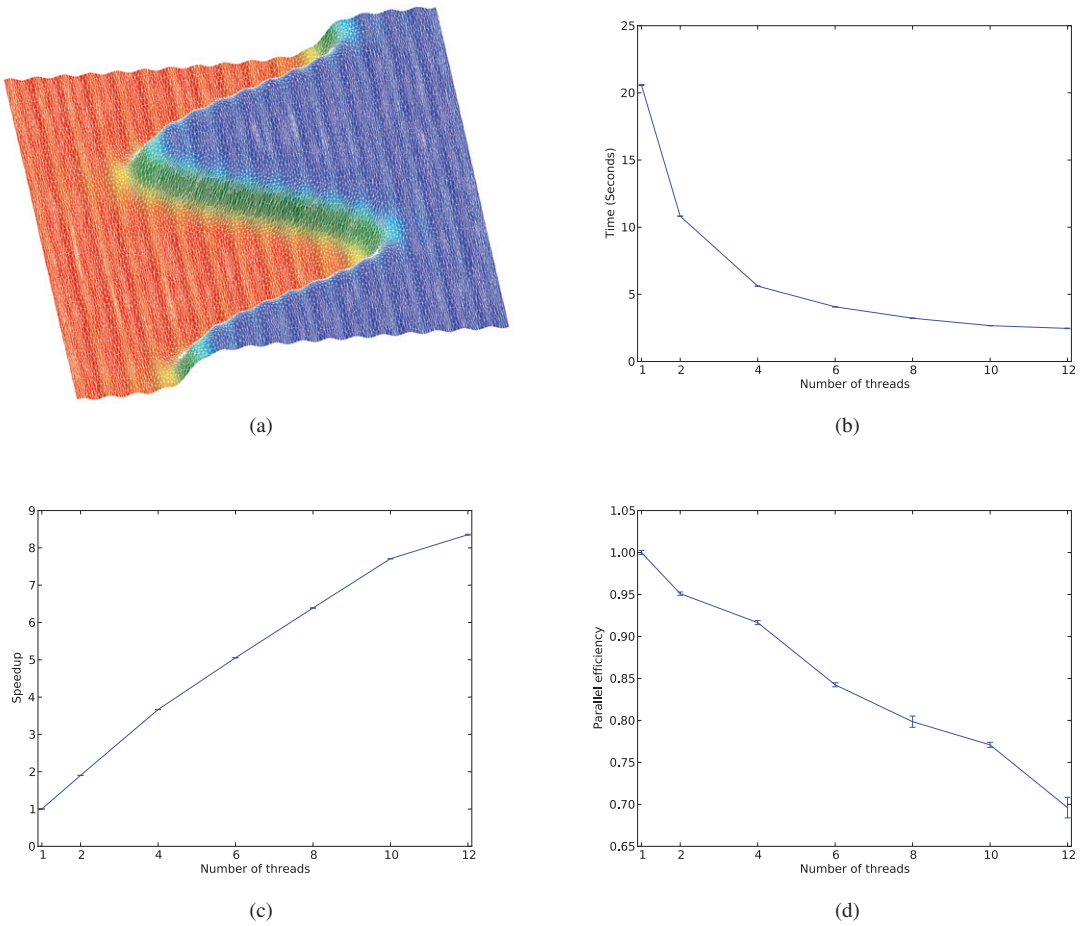


Figure 3: The figure shows (a) the mesh and benchmark function; (b) execution time when using between 1 and 12 cores; (c) the resulting speedup; and (d) the parallel efficiency.

The benchmark function is taken from [22]:

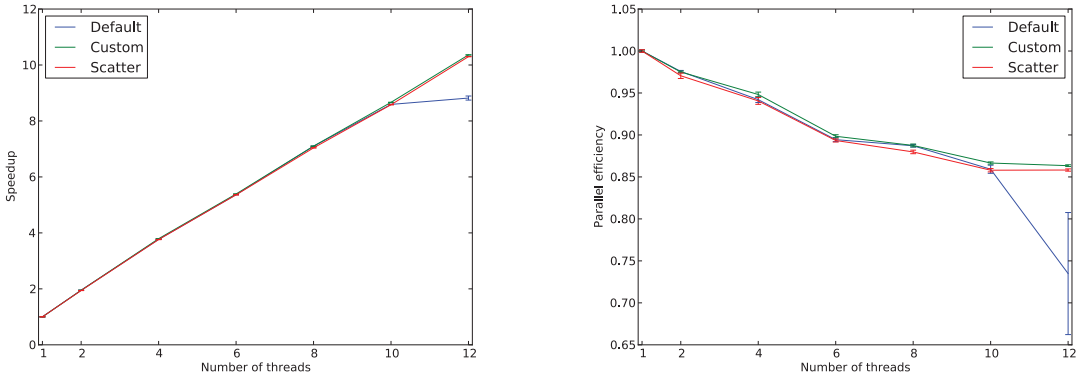
$$\forall (x, y) \in [0, 1]^2, f(x, y) = 0.1 \sin(50x) + \arctan\left(\frac{0.1}{\sin(5y) - 2x}\right). \tag{9}$$

The high amplitude shock is induced by the arctan function and small variations in amplitude are superimposed with a sine function, see Figure 3(a). The L1 error norm was used to form the metric tensor field as it is more effective at capturing multiscale features [22].

The mesh used in these numerical experiments has 9958 vertices and 19560 elements. It was generated by initially adapting the mesh using anisotropic mesh modification; specifically, applying mesh refinement, coarsening and swapping to satisfy the metric tensor field defined above [23]. Table 1 gives the mean, minimum, root-mean-squared values for element qualities using the quality measure defined in equation (3), for the initial mesh and for the different smoothing algorithms tested. The time taken for the smoothing to converge is also given when 12 threads are used. While the mean quality is important, arguably most attention needs to be paid to the minimum quality as this is where the highest discretisation errors will be incurred. As can be seen from Table 1, application of the smart Laplacian already significantly improves the minimum element quality from the initial state at a very low computational cost. When this is combined with a line search to locate a local maximum, the mean does not vary significantly, but there is

Table 1: Improvements to quality metrics for different smoothing approaches. The time taken figure is for the case where 12 threads are used along with the parallel harness from Algorithm 4.

	Mean quality	Minimum quality	RMS quality	Time (seconds)
Initial	0.76	0.03	0.16	-
Algorithm 1: Smart Laplacian	0.82	0.18	0.13	0.06
Algorithm 1 + line search	0.83	0.26	0.13	0.16
Algorithm 2: L_∞ optimisation	0.81	0.41	0.12	1.38



(a) (b)

Figure 4: The figure shows (a) the resulting speedup; and (b) the parallel efficiency for Algorithm 2 + Algorithm 3. *Default* indicates no thread/core affinity was set; *Custom* indicates that a thread/core was designed to lump adjacent data into the same NUMA memory node; *Scatter* enforces an affinity which tries to evenly distribute threads throughout the NUMA region but is not aware of data relationships.

a marked improvement in the minimum element quality in the mesh. L_∞ optimisation provides the greatest improvement in mesh quality as defined by the minimum element quality. However, the computational cost is significantly greater than the next best method.

As can be observed in Figure 4, data affinity can have a serious impact on the scaling of an algorithm, particularly when all the cores are used and the algorithm peaks in its consumption of memory bandwidth. We can also see from the error bars that when threads are not pinned to cores there can be significant variance in runtime. Again, this is only really apparent when all the cores are being exploited. The best run time in this experiment was 9.9 seconds when using 12 threads. Therefore, comparing the results in Table 1 it is clear that in this case that the dynamic schedule is more appropriate as the number of active vertices is reduced. More investigation is required to determine if this is always the case or if some run-time decision making is required to switch between the two approaches.

Figure 3(b, c, d) shows the strong scaling results of L_∞ optimisation using 1-12 OpenMP threads. Importantly, we continue to get good parallel efficiency right out to the total number of cores. The main reason for the drop off in parallel efficiency is that the quantity of the mesh which is optimised decreases as the algorithm progresses, therefore the number of kernels to be executed in an iteration over the mesh becomes small relative to the number of execution threads.

5. Conclusion

Optimisation based anisotropic mesh smoothing is highly effective at raising the quality of the poorest elements of an unstructured mesh. As it is gradient based, it is a much more direct approach to improving element quality than

heuristic methods commonly used, such as variations on Laplacian smoothing.

Using colouring based methods it is possible to achieve high degrees of concurrency and excellent fine grained scaling behaviour. While for some variations of the algorithm data locality is clearly a key issue; progressively reducing the amount of data that needs to be operated upon at each iteration means that the algorithm needs to switch to a dynamic schedule.

Future work will focus on evaluating these strategies for co-processors and to experiment with different renumbering schemes to improve cache coherency between elements in the same independent set.

Acknowledgements

The authors would like to thank Fujitsu Laboratories of Europe Ltd. and EPSRC grant no. EP/I00677X/1 for supporting this work. ICT-HPC provided the computational resources.

References

[1] C. Pain, M. Piggott, A. Goddard, F. Fang, G. Gorman, D. Marshall, M. Eaton, P. Power, C. De Oliveira, Three-dimensional unstructured mesh ocean modelling, Ocean Modelling 10 (1-2) (2005) 5–33.

[2] J. Southern, G. Gorman, M. Piggott, P. Farrell, Parallel anisotropic mesh adaptivity with dynamic load balancing for cardiac electrophysiology, Journal of Computational Science.

[3] J. Dongarra, What you can expect from exascale computing, in: International Supercomputing Conference (ISC'11), Hamburg, Germany, 2011.

[4] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on, IEEE, 2009, pp. 427–436.

[5] D. A. Field, Laplacian smoothing and Delaunay triangulations, Communications in Applied Numerical Methods 4 (1988) 709—712.

[6] V. N. Parthasarathy, S. Kodiyalam, A constrained optimization approach to finite element mesh smoothing, Finite Element in Analysis and Design 9 (4) (1991) 309—320.

[7] S. A. Canann, M. B. Stephenson, T. Blacker, Optismoothing: An optimization-driven approach to mesh smoothing, Finite Elements in Analysis and Design 13 (1993) 185–190.

[8] L. Freitag, C. Ollivier-Gooch, A comparison of tetrahedral mesh improvement techniques (1996).

[9] L. A. Freitag, C. Ollivier-Gooch, Tetrahedral mesh improvement using swapping and smoothing, International Journal for Numerical Methods in Engineering 40 (1997) 3979—4002.

[10] N. Amenta, M. Bern, D. Eppstein, Optimal point placement for mesh smoothing, in: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1997.

[11] S. Canann, J. Tristano, M. Staten, An approach to combined Laplacian and optimization-based smoothing for triangular, quadrilateral, and quad-dominant meshes, in: Proceedings, 7th International Meshing Roundtable, 1998, pp. 479—494.

[12] L. A. Freitag, C. Ollivier-Gooch, Tetrahedral mesh improvement using swapping and smoothing, International Journal for Numererical Methods in Engineering 40 (21) (1997) 3979—4002.

[13] C. C. Pain, A. P. Umpleby, C. R. E. de Oliveira, A. J. H. Goddard, Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations, Computer Methods in Applied Mechanics and Engineering 190 (29-30) (2001) 3771–3796.

[14] L. Freitag, M. Jones, P. Plassmann, An efficient parallel algorithm for mesh smoothing, in: Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories, Citeseer, 1995, pp. 47–58.

[15] P. Knupp, Achieving finite element mesh quality via optimization of the Jacobian matrix norm and associated quantities. Part I — A framework for surface mesh optimization, International Journal for Numerical Methods in Engineering 48 (3) (2000) 401–420.

[16] P. Knupp, Achieving finite element mesh quality via optimization of the Jacobian matrix norm and associated quantities. Part II — A framework for volume mesh optimization and the condition number of the Jacobian matrix, International Journal for Numerical Methods in Engineering 48 (8) (2000) 1165–1185.

[17] Y. Vasilevskii, K. Lipnikov, An adaptive algorithm for quasioptimal mesh generation, Computational mathematics and mathematical physics 39 (9) (1999) 1468–1486.

[18] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, D. Stanzione, Early experiences with the Intel many integrated cores accelerated computing technology, in: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, ACM, 2011, p. 21.

[19] E. Boman, D. Bozdağ, U. Catalyurek, A. Gebremedhin, F. Manne, A scalable parallel graph coloring algorithm for distributed memory computers, Euro-Par 2005 Parallel Processing (2005) 613–613.

[20] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan data management services for parallel dynamic applications, Computing in Science and Engineering 4 (2) (2002) 90–97.

[21] G. Karypis, V. Kumar, MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, `http://www.cs.umn.edu/ metis` (2009).

[22] A. Loseille, F. Alauzet, Optimal 3D highly anisotropic mesh adaptation based on the continuous mesh framework, Proceedings of the 18th International Meshing Roundtable (2009) 575–594.

[23] P. George, F. Hecht, E. Saltel, Automatic mesh generator with specified boundary, Computer methods in applied mechanics and engineering 92 (3) (1991) 269–288.