Thread Parallelism for Highly Irregular Computation in Anisotropic Mesh Adaptation

Georgios Rokos
Imperial College London
London, United Kingdom
georgios.rokos09@imperial.ac.uk

Kristian Ejlebjerg Jensen
Imperial College London
London, United Kingdom
kristianejlebjerg@gmail.com

Gerard J. Gorman
Imperial College London
London, United Kingdom
g.gorman@imperial.ac.uk

Paul H. J. Kelly
Imperial College London
London, United Kingdom
p.kelly@imperial.ac.uk

ABSTRACT

Thread-level parallelism in irregular applications with mutable data dependencies presents challenges because the underlying data is extensively modified during execution of the algorithm and a high degree of parallelism must be realized while keeping the code race-free. In this article we describe a methodology for exploiting thread parallelism for a class of graph-mutating worklist algorithms, which guarantees safe parallel execution via processing in rounds of independent sets and using a deferred update strategy to commit changes in the underlying data structures. Scalability is assisted by atomic fetch-and-add operations to create worklists and work-stealing to balance the shared-memory workload. This work is motivated by mesh adaptation algorithms, for which we show a parallel efficiency of 60% and 50% on Intel®Xeon® Sandy Bridge and AMD Opteron™ Magny-Cours systems, respectively, using these techniques.

Keywords

anisotropic mesh adaptivity, irregular data, shared-memory parallelism, manycore, parallel worklist algorithm, topology mutation, graph colouring, work-stealing, deferred update

1. INTRODUCTION

Finite element/volume methods (FEM/FVM) are commonly used in the numerical solution of partial differential equations (PDEs). Unstructured meshes, where the spatial domain has been discretised into simplices (*i.e.* triangles in 2D, tetrahedra in 3D), are of particular interest in applications where the geometric domain is complex and structured meshes are not practical. Simplices are well suited to varying mesh resolution throughout the domain, allowing for local coarsening and refinement of the mesh without hanging nodes. On the other hand, this flexibility introduces compli-

cations of its own, such as management of mesh quality and computational overheads arising from indirect addressing.

Computational mesh resolution is often the limiting factor in simulation accuracy. 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 (*e.g.* [15, 20]). A difficulty with mesh-based modelling is that the mesh is generated before the solution is known, however, the local solution error is related to the local mesh resolution. Overly coarse meshes lead to low accuracy whereas over-refined meshes can greatly increase the computational cost.

Mesh adaptation methods provide an important means to minimise computational cost while still achieving the required accuracy [16, 12]. 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 compute a solution to a specified error tolerance while using the minimum resolution everywhere in the domain and maintaining element quality constraints.

Previous work has described how adaptive mesh methods can be parallelised for distributed-memory systems using MPI (*e.g.* [12, 10]). However, there is a continuous trend towards an increasing number of cores per compute node in the world's most powerful supercomputers and it is assumed that the nodes of a future exascale system will each contain thousands of cores [7]. Therefore, it is important that algorithms are developed with very high levels of parallelism and using thread-parallel programming models, such as OpenMP [5], that exploit the memory hierarchy. However, irregular applications are hard to parallelise effectively on shared-memory architectures for reasons described in [13].

In this article we take a fresh look at anisotropic adaptive mesh methods and parallelise them using new scalable techniques suitable for modern multicore and manycore architectures. These concepts have been implemented in the open source framework PRAgMaTIc (Parallel anisotRopic Adaptive Mesh ToolkIt)[1]. The remainder of the paper is laid out as follows: §2 gives an overview of the mesh adaptation pro-

---

[1]http://meshadaptation.github.io/

cedure; §3 describes the new irregular compute methodology used to parallelise the adaptive algorithms; and §4 illustrates how well our framework performs in 2D and 3D benchmarks. We conclude the paper in §5.

2. MESH ADAPTIVITY BACKGROUND

In this section we give an overview of anisotropic mesh adaptation, focusing on the element quality as defined by an error metric and the adaptation kernels which iteratively improve 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 (a) solution errors may be unacceptably high and (b) parts of the solution may be over-resolved, thereby incurring unnecessary computational expense. A solution to this is to compute appropriate local error estimates and use them to dynamically control the local mesh resolution at runtime. 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 [11].

2.2 Element quality

As discretisation errors are dependent upon element shape as well as size, a number of measures of element quality have been proposed, out of which, in the work described here, we use the quality functionals by Vasilevskii *et al.* for triangles [22] and tetrahedra [1], which indicate that the ideal element is an equilateral triangle/tetrahedron with edges of unit length measured in metric space.

2.3 Overall adaptation procedure

The mesh is adapted through a series of local operations: edge collapse, edge refinement, element-edge swaps and vertex smoothing. While the first two of these operations control the local resolution, the latter two are used to improve the element quality. Algorithm 1 gives a high level view of the anisotropic mesh adaptation procedure as described by Li *et al.* [12]. The inputs are $\mathcal{M}$, the piecewise linear mesh from the modelling software, and $\mathcal{S}$, the node-wise metric tensor field which specifies anisotropically the local mesh resolution requirements. The process involves the iterative application of coarsening, swapping and refinement to optimise the resolution and quality of the mesh. The loop 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, which aims primarily at improving the worst-element quality. Smoothing is left out of the main loop because it is an expensive operation and it is found empirically that it is efficient to fine-tune the worst-element quality once mesh topology has been fixed.

2.4 Adaptation kernels

A brief description of the four mesh optimisation kernels follows. Figure 1 shows 2D examples to demonstrate what each kernel does to the local mesh patch, but the same operations are applied in an identical manner in 3D. For more

---

Algorithm 1 Mesh optimisation procedure.

Inputs: $\mathcal{M}$, $\mathcal{S}$.
repeat
$\quad (\mathcal{M}^*, \mathcal{S}^*) \leftarrow coarsen(\mathcal{M}^*, \mathcal{S}^*)$
$\quad (\mathcal{M}^*, \mathcal{S}^*) \leftarrow swap(\mathcal{M}^*, \mathcal{S}^*)$
$\quad (\mathcal{M}^*, \mathcal{S}^*) \leftarrow refine(\mathcal{M}^*, \mathcal{S}^*)$
until (max. number of iterations **or** convergence)
$(\mathcal{M}^*, \mathcal{S}^*) \leftarrow smooth(\mathcal{M}^*, \mathcal{S}^*)$
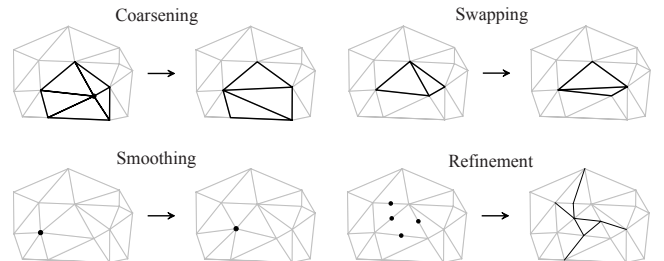**return** $\mathcal{M}^*$

Figure 1: Examples of the four adaptive kernels.

details on the adaptive algorithms the reader is referred to the publications by Li *et al.* [12] (coarsening, refinement, swapping) and Freitag *et al.* [8, 9] (smoothing).

2.4.1 Coarsening

Coarsening is the process of lowering mesh resolution locally by collapsing an edge to a single vertex, thereby removing all elements that contain this edge, leading to a reduction in the computational cost.

2.4.2 Refinement

Refinement is the process of increasing mesh resolution locally by (a) splitting of edges which are longer than desired (as indicated by the error estimation) and (b) subsequent division of elements using refinement templates [6].

2.4.3 Swapping

Swapping is done in the form of flipping an edge shared by two elements, considering the quality of the swapped elements - if the minimum quality is improved then the original mesh elements are replaced with the edge-swapped elements.

2.4.4 Smoothing

The kernel of vertex smoothing relocates a central vertex so that the local mesh quality is increased. Common heuristic methods are the quality-constrained Laplacian smoothing [8] and the more expensive optimisation-based smoothing [9].

2.4.5 Propagation

The operations of coarsening, swapping and smoothing often need to be propagated to the local mesh neighbourhood. When a kernel is applied onto an edge/vertex, neighbouring edges/vertices need to be reconsidered for processing because the topological/geometrical changes that occurred might give rise to new configurations of better quality. Therefore, these adaptive algorithms keep sweeping over the mesh until no further changes are made.

3. IRREGULAR COMPUTE METHOD

To allow fine grained parallelisation of mesh adaptation we based our methodology on graph colouring, following a proposal by Freitag *et al.* [10]. 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 update strategy, described below, to minimise synchronisations and allow parallel writes. Additionally, we make use of atomic operations to create parallel worklists in a synchronisation-free fashion and, finally, try to balance the workload among threads using work-stealing [3].

3.1 Hazards

There are two types of hazards when running mesh optimisation algorithms in parallel: *topological hazards*; and *data races*. The former refers to the situation where an adaptive operation results in invalid or non-conforming edges and elements. For example in coarsening, if some vertex V_B collapses onto another vertex V_A, then V_A cannot collapse onto some other vertex at the same time. Data races can occur when two threads try to update the same adjacency list of a vertex concurrently. For example in coarsening, two neighbours of some vertex V_A can collapse onto V_A concurrently, then V_A's adjacency list has to be updated to reflect the changes made by the coarsening operations. Concurrent access to V_A's adjacency list may lead to race conditions.

3.2 Colouring

Topological 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. The adaptive algorithm then processes the mesh in batches of independent sets. The fact that topological changes are made to the mesh means that colouring is invalidated frequently and the mesh has to be re-coloured before proceeding to the next iteration of the adaptive algorithm. Therefore, we need to use a fast and scalable colouring algorithm (see [18]).

3.3 Deferred Update

Colouring does not eliminate data races when updating adjacency lists. A 2-distance colouring was not considered here as it is expensive and increases the chromatic number, effectively reducing the exposed parallelism. Instead, in a shared-memory environment with N threads, each thread allocates a private collection of N lists. When the adjacency list of some vertex V_i has to be updated, the thread executing the adaptive kernel does not commit the update immediately; instead, it pushes the operation back into the list for thread $tid = ID(V_i)\%N$, where $ID(V_i)$ is the integer identifier of V_i. After processing an independent set and before proceeding to the next one, every thread T_i visits the private collections of all threads, locates the list reserved for T_i and commits the updates stored there. This way, it is guaranteed that one and only thread will update the adjacency list of any given vertex. We call this technique the *deferred update*. Code Snippet 1 demonstrates a typical usage scenario. An important advantage of this strategy is that we always read the most up-to-date data when executing an adaptive kernel (as if we used an "as we go" write-back scheme), eliminating the risk of mesh data corruption in coarsening, refinement and swapping and having a faster-converging Gauss-Seidel-style iteration process in smoothing.

```
1  typedef std::vector<Updates> DefUpdList;
2  int N = omp_get_max_threads();
3
4  // N collections of deferred-update lists
5  std::vector< std::vector<DefUpdList> > defUpd(N);
6
7  #pragma omp parallel
8  {
9    int tid = omp_get_thread_num();
10   // Allocate one list for each thread.
11   defUpd[tid].resize(N);
12
13   // Process the independent set in parallel
14   #pragma omp for
15   for(int i=0; i<nVerticesInSet; ++i){
16     update = execute_kernel(i);
17     // To be committed by thread i%N.
18     defUpd[tid][i%N].push_back(update);
19   }
20
21   // Commit updates tid is responsible for.
22   for(int i=0; i<N; ++i)
23     commit_all_updates(defUpd[i][tid]);
24
25   // Proceed to the next independent set...
26 }
```

Code Snippet 1: Example of the deferred update scheme.

3.4 Worklists and Atomic-Capture

There are many cases where it is necessary to create a worklist of items which need to be processed, *e.g.* for propagation of adaptive operations. New work items generated locally by a thread need to be accumulated into a global worklist over which the next invocation of the adaptive kernel will iterate. The classic approach based on prefix sums [2] requires thread synchronisation and was found limiting in terms of scalability. A better method is based on atomic fetch-and-add on a global integer which stores the size of the worklist needed so far. Every thread increments this integer atomically while caching the old value. This way, the thread knows where to copy its private data and increments the integer by the size of this data, so the next thread to access the integer knows in turn where to copy its private data. An example of using this technique via OpenMP's atomic-capture clause [14] is given in Code Snippet 2, where it is shown that no thread synchronisation is needed to generate the global worklist (note the `nowait` clause). The overhead/spinlock associated with atomic-capture operations was found to be insignificant.

3.5 Work-stealing

Work-stealing [3] is a sophisticated technique aiming at balancing workload among threads while keeping scheduling overhead as low as possible. For-loop scheduling strategies provided by the OpenMP runtime system were found to be inadequate, either incurring significant scheduling overhead or leading to load imbalances. As of version 4.0, OpenMP does not support work-stealing for parallel for-loops so we created a novel scheduler [19] which differs from other proposals in two ways: it engages a heuristic to help the thief find a suitable victim to steal from; and uses POSIX signals/interrupts to accomplish stealing in an efficient manner.

4. EXPERIMENTAL RESULTS

We will show adaptivity results for viscous fingering in 2D and structural compliance minimisation in 2D and 3D, followed by performance evaluation.

```cpp
int worklistSize = 0;
std::vector<Item> globalWorklist(prealloc_size);

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

  #pragma omp for nowait
  for(all items which need to be processed){
    new_item = do_some_work();
    private_list.push_back(new_item);
  } // No need to synchronise at end of loop.

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

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

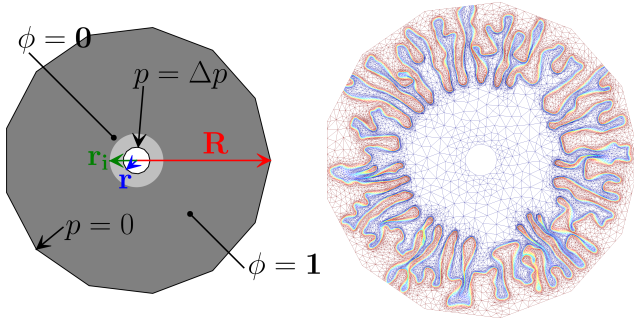Code Snippet 2: Creating a worklist using atomic-capture.

Figure 2: The initial condition for the viscous fingering (left) and a snapshot of a simulation (right).

4.1 Viscous Fingering

Viscous fingering is a limiting process for enhanced oil recovery technologies. It happens whenever one fluid displaces another with a higher viscosity [17], typically in a porous media. A typical setup and simulation is shown in Figure 2, with the blue fluid having a viscosity e^2 times lower than the red fluid. The initial saturation is unperturbed and it is thus the length scale of the initial mesh that triggers the instability. Mesh adaptation is driven by the Hessian of the pressure combined with the Hessian of the saturation ϕ [4].

4.2 Structural Optimisation

Structural compliance minimisation is concerned with the problem of finding stiff and lightweight mechanical components [21], often in the context of linear elasticity. The setup for a classical cantilever problem with support to the left and a load to the right is shown in Figure 3 (top left). The question is how to form the stiffest possible link between the two boundaries, given a certain amount of isotropic material. The problem is ill-posed unless a minimum length scale is imposed for the design, because the optimal structure is a composite. In fact, one can see a tendency towards microstructured areas when a small minimum length scale $L_{min} = 10^{-3}L_{char}$ is used as illustrated in Figure 3 (bottom left). Note how the many straight and parallel connections
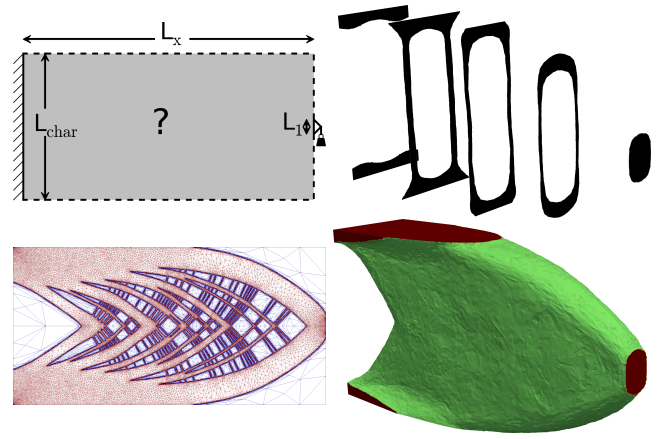

Figure 3: The setup for structural compliance minimisation (top left) and the result for the case of a small minimum length scale (bottom left). Red corresponds to solid areas and blue to void. The result of compliance minimisation in 3D is shown in terms of the cross-sectional view (top right) and the solid/void interface (bottom right).

can be efficiently resolved with anisotropic elements. Mesh adaptation is driven by the Hessians of the design and the topological derivative [21]. A Helmholtz filter is applied to both design and derivative to smooth out features smaller than L_{min}, before the Hessians are computed.

The two dimensional setup is also extruded to three dimensions, as plotted in Figure 3 (top right and bottom right). Note that the large planar areas with little curvature are well resolved by the anisotropic elements. The increased dimensionality leads to a much simpler topology even though the optimisation is performed with $L_{min} = 5 \cdot 10^{-3}L_{char}$.

In order to evaluate the parallel performance, a synthetic solution ψ is defined to vary in time and space:

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

where T is the period. This is a good choice as a benchmark as it contains multi-scale features and a shock front, *i.e.* the typical solution characteristics where anisotropic adaptive mesh methods excel. An isotropic mesh was generated on the unit square using approximately 200×200 triangles and the adaptation benchmark was run with a requirement for $\approx 550k$ elements. The same example was extruded in 3D, where an isotropic mesh was generated in the unit cube using approximately $50 \times 50 \times 50$ tetrahedra and the adaptation benchmark was run with a requirement for $\approx 210k$ elements. 3D swapping has not been parallelised, therefore the corresponding results have been omitted.

The code was compiled using the Intel compiler suite (version 14.0.1) and with the `-O3` optimisation flag. We used two systems to evaluate performance: (a) a dual-socket Intel® Xeon® E5-2650 system (Sandy Bridge, 2.00GHz, 8 cores per socket, 2-way hyper-threading) running Red Hat®Enterprise Linux® Server release 6.4 (Santiago) and (b) a quad-socket AMD Opteron™ 6176 system (Magny-Cours, 2.3GHz, 12 cores per socket) running Ubuntu 12.04.5. In all cases,

thread-core affinity was used. Figures 4 and 5 show the average (over 50 time steps) execution time per time step and parallel efficiency against the number of threads using single-socket (SS), dual-socket (DS) and quad-socket (QS) configurations. On the Intel®Xeon® system we also enable hyper-threading (HT) to make use of all 40 logical cores.

Running on one socket, our code achieves a parallel efficiency of over 60% on Intel®Xeon® and around 50% on AMD Opteron™. Smoothing scales better than the other algorithms as it is more compute-intensive, which favours scalability, reaching an efficiency of over 50% even in the quad-socket case. When we move to more sockets, NUMA effects become pronounced, which is expected as common NUMA optimisations such as pinning and first-touch for page binding are ineffective for irregular computations. Nonetheless, the achievable efficiency is good considering the irregular nature of those algorithms.

5. CONCLUSION

In this paper we examined the scalability of anisotropic mesh adaptivity using a thread-parallel programming model and explored 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 using a combination of a fast graph colouring technique, the deferred-update strategy, atomic-based creation of worklists and for-loop work-stealing. In principle, this methodology facilitates scaling up to the point where the number of elements of an independent set is equal to the number of available threads.

Acknowledgments

6. REFERENCES

[1] A. Agouzal, K. Lipnikov, and Y. Vassilevski. Adaptive generation of quasi-optimal tetrahedral meshes. *East-West Journal*, 7(4):223–244, 1999.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[4] L. Chen, P. Sun, and J. Xu. Optimal anisotropic meshes for minimizing interpolation errors in L^p-norm. *Mathematics of Computation*, 76(257):179–204, 2007.

[5] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[6] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.
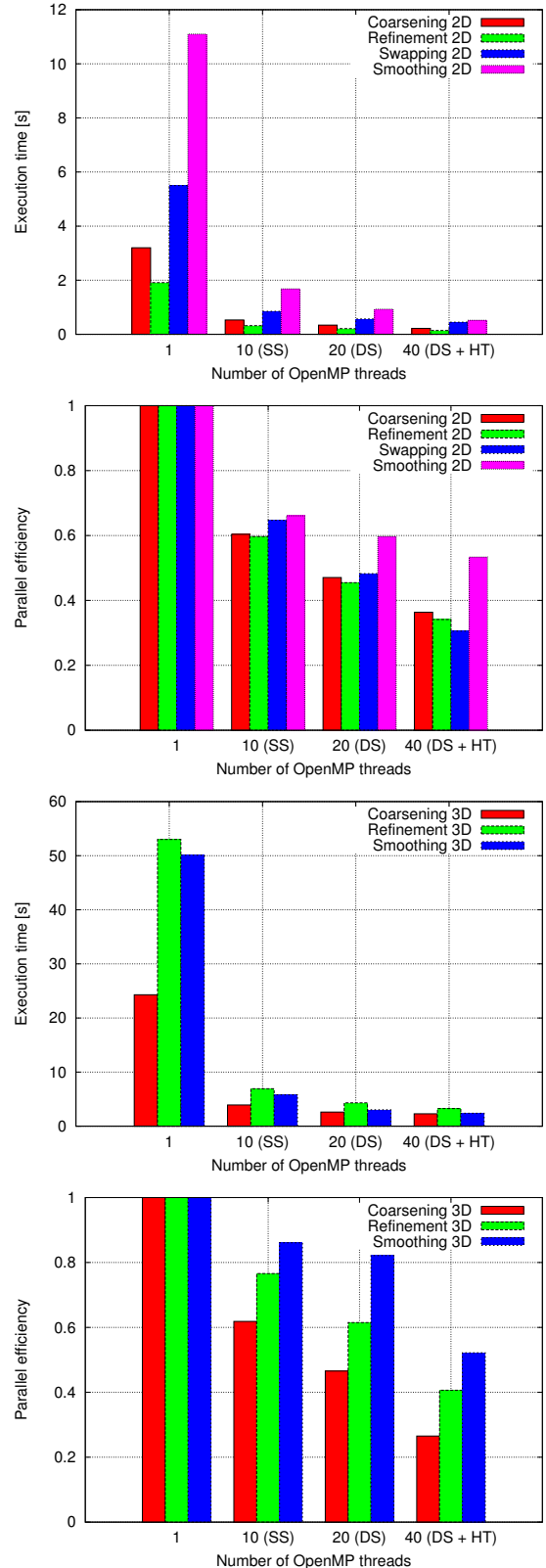
Figure 4: Execution time and parallel efficiency of 2D and 3D synthetic benchmarks on the 2x8-core Intel®Xeon® Sandy Bridge system.
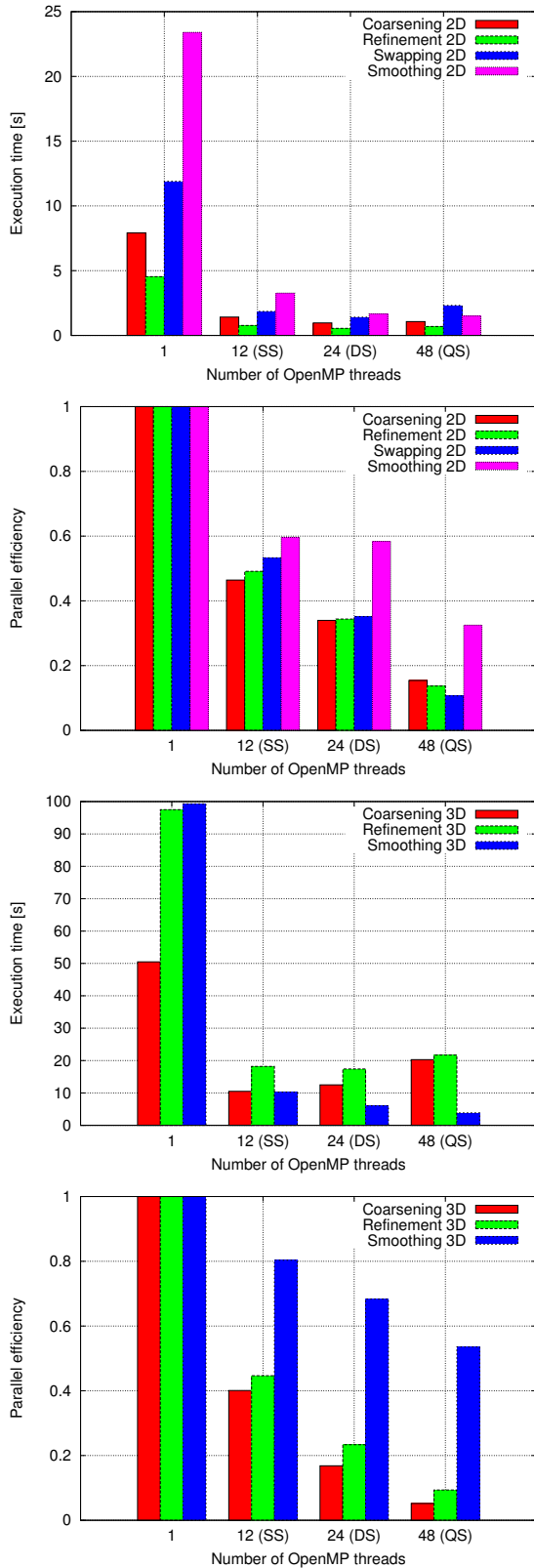
Figure 5: Execution time and parallel efficiency of 2D and 3D synthetic benchmarks on the 4x12-core AMD Opteron™ Magny-Cours system.

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

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

[9] L. A. Freitag and P. M. Knupp. Tetrahedral mesh improvement via optimization of the element condition number. *International Journal for Numerical Methods in Engineering*, 53(6):1377–1391, 2002.

[10] L. F. Freitag, M. T. Jones, and P. E. Plassmann. The Scalability Of Mesh Improvement Algorithms. In *IMA Volumes in Mathematics and its Applications*, pages 185–212. Springer-Verlag, 1998.

[11] P.-J. Frey and F. Alauzet. Anisotropic mesh adaptation for cfd computations. *Computer methods in applied mechanics and engineering*, 194(48):5068–5082, 2005.

[12] X. Li, M. Shephard, and M. Beall. 3d anisotropic mesh adaptation by mesh modification. *Computer methods in applied mechanics and engineering*, 194(48-49):4915–4950, 2005.

[13] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[14] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, July 2013.

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

[16] M. D. Piggott, P. E. Farrell, C. R. Wilson, G. J. Gorman, and C. C. Pain. Anisotropic mesh adaptivity for multi-scale ocean modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4591–4611, 2009.

[17] S. Pramanik, G. Kulukuru, and M. Mishra. Miscible viscous fingering: Application in chromatographic columns and aquifers. In *COMSOL conference, Bangalore*, 2012.

[18] G. Rokos, G. J. Gorman, and P. H. J. Kelly. A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures. *pre-print*, 2015. http://arxiv.org/abs/1505.04086.

[19] G. Rokos, G. J. Gorman, and P. H. J. Kelly. An Interrupt-Driven Work-Sharing For-Loop Scheduler. *pre-print*, 2015. http://arxiv.org/abs/1505.04134.

[20] J. Southern, G. Gorman, M. Piggott, and P. Farrell. Parallel anisotropic mesh adaptivity with dynamic load balancing for cardiac electrophysiology. *Journal of Computational Science*, 2011.

[21] K. Suresh. A 199-line matlab code for pareto-optimal tracing in topology optimization. *Structural and Multidisciplinary Optimization*, 42(5):665–679, 2010.

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