有纸质笔记

Anisotropic Mesh Adaptation for the Manycore Era

Georgios Rokos

Imperial College London

April 23, 2015

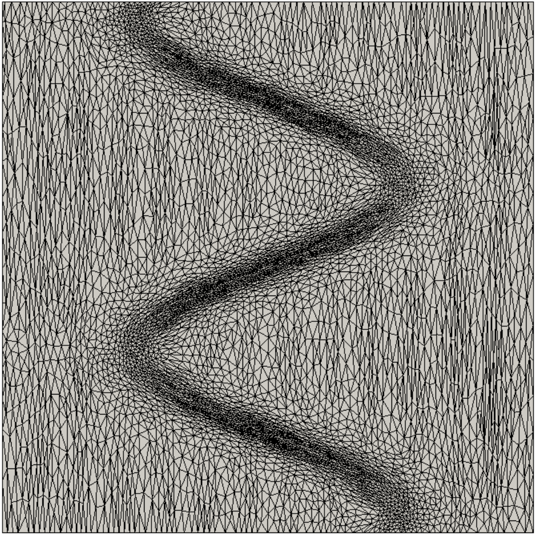
Introduction

- Irregular applications:
  - Use of unstructured or completely irregular data
  - Mostly represented as graphs
- Challenges:
  - Unpredictable memory access patterns
  - Poor data locality
  - Kernels end up being memory-bound rather than compute-bound
  - Data-driven algorithms, hard to extract parallelism
  - Fine-grained parallelism leads to frequent thread synchronisation
- Mutable dependencies:
  - E.g. *morph algorithms* (Pingali *et al.*)
  - Graph topology is mutated in non-trivial ways
  - Any preprocessing is constantly invalidated

Mesh adaptivty

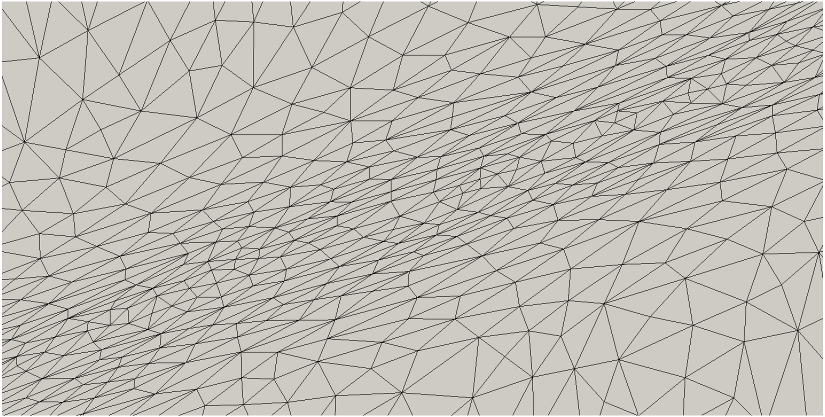
- Need to study a real-world problem in order to develop techniques for parallelising irregular kernels
- Unstructured meshes and finite element/volume modelling:
    - Spatial domain discretised into triangles (in this talk we only focus on 2D)
    - Ideal for representing complex geometries (e.g. coastal modelling)
    - Numerical solutions of partial differential equations (PDEs)
- Mesh adaptivity methods:
    - Allow dynamic control of solution error
    - Keep the resolution in the goldilocks zone - not too high and not too low
    - Minimise computational cost for a specific model accuracy

Example



Example: Detail along the wave front

- Elements are stretched along the direction of the front



Error control

- Initial mesh generated *a priori*:
  - Difficult to generate a mesh that is both efficient and resolves the solution where required
  - Particularly difficult for multi-scale problems
- Local error estimates
  - Error estimate transformed to a metric tensor field (MTF)
  - Discretised vertex-wise
  - Tensor at some vertex specifies local size and shape of an element containing that vertex which is required to achieve a specific error tolerance
- Support for anisotropic problems
  - PDE exhibits directional dependencies (desired element size and shape) encoded in a MTF
  - E.g. higher resolution is required perpendicular to a shock front (where flow is more complex) than along the shock

Metric tensor

- A metric tensor is a symmetric matrix, 2x2 in 2D, 3x3 in 3D
- Defines length of vectors
- Allows us to calculate inner products in generalised spaces, in the same way the dot product defines distance in Euclidean space
- Example in 2D with vertices $V_1(x_1, y_1)$, $V_2(x_2, y_2)$ and edge $\mathbf{E} = (x_0, y_0) = (x_2 - x_1, y_2 - y_1)$
  - Length in Euclidean space given by the dot product:

$$L_{Euclidean} = \parallel \mathbf{E} \parallel = \sqrt{\mathbf{E} \cdot \mathbf{E}} = \sqrt{x_0^2 + y_0^2}$$

  - Edge length with respect to a metric tensor $\mathbf{M} = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$:

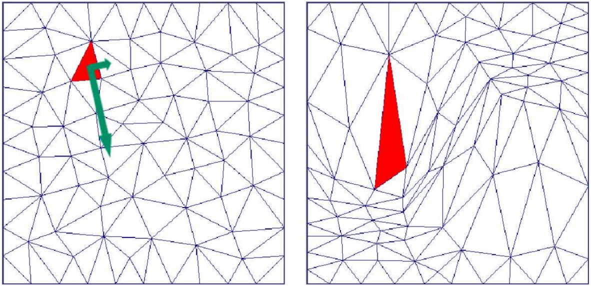
$$L_M = \parallel \mathbf{E} \parallel_M = \sqrt{\mathbf{E}^T \mathbf{M} \mathbf{E}} = \sqrt{\begin{bmatrix} x_0 y_0 \end{bmatrix} \begin{bmatrix} A & B \\ B & C \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}} = \tag{1}$$
$$= \sqrt{x_0^2 A + 2 x_0 y_0 B + y_0^2 C}$$

Element size and shape

- Metric tensor in the middle of a triangle
  - Linear interpolation of metric tensors at the three vertices
- Eigenvalue decomposition of a 2D metric tensor:

$$\mathbf{M} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T = \begin{bmatrix} Q_{00} & Q_{01} \\ Q_{10} & Q_{11} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \end{bmatrix} \begin{bmatrix} Q_{00} & Q_{10} \\ Q_{01} & Q_{11} \end{bmatrix}$$

- Each eigenvalue λ_i encodes the required element size in the direction of the corresponding eigenvector Q_i



Adaptive algorithms

- 4 adaptive algorithms
  - Coarsening
  - Refinement
  - Swapping

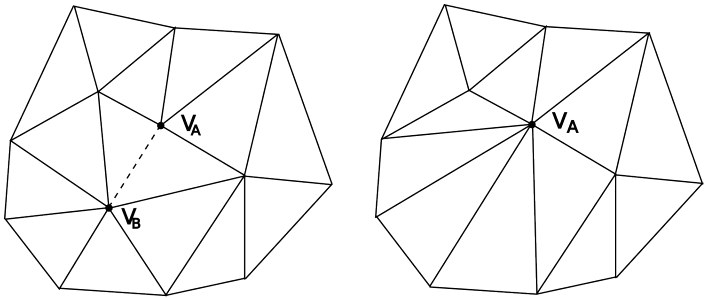
  $\left.\right\}$ h-adaptivity $\longrightarrow$ mesh topology is modified

  - Smoothing $\left.\right\}$ r-adaptivity $\longrightarrow$ mesh topology is not modified

- Mesh adaptation
  - Element quality functional measures 'distance' from ideal element as defined by metric field
  - Iterative application of local mesh operations until the quality is within some threshold
  - h-adaptivity: morph algorithms

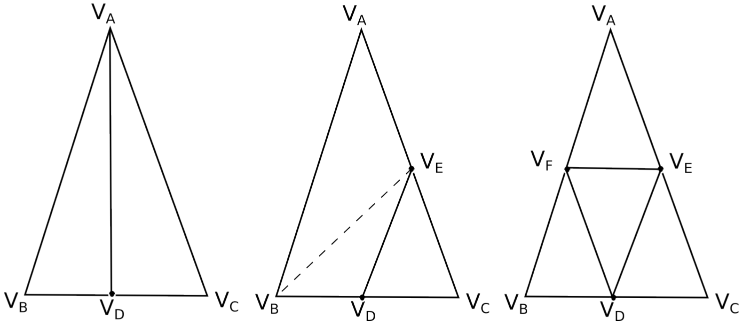
Coarsening

- Done via edge collapse: vertex V_B collapses onto V_A, removing the dashed edge and the adjacent elements from the mesh (Li et al. 2005)
- Every vertex is examined to determine onto which neighbour (if any) it can collapse
- If a vertex is removed the local neighbourhood is modified, so all neighbours are marked for re-examination
  $\Longrightarrow$ Propagation of coarsening



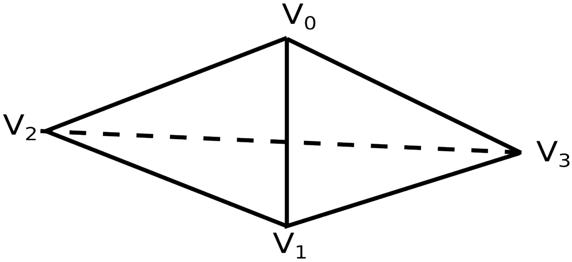
Refinement

- Edge and element refinement: long edges are split, leading to 1:2 (bisection), 1:3 or 1:4 (regular refinement) division of elements, which increases local mesh resolution (Li et al. 2005)
- At first, all edges are visited and long edges are split
- Next up, elements with a split edge are split according to the number of split edges
- No need for propagation, just execute refinement kernel again



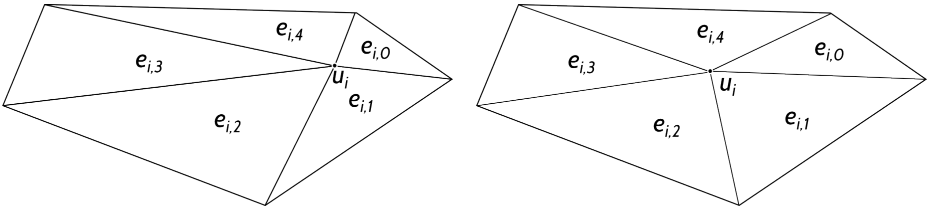
Swapping

- Edge swapping: edges shared between two elements can be flipped if the minimum quality of the element pair is raised (Li et al. 2005)
- Improves mesh quality without increasing the number of elements
- Once an edge has been flipped, all adjacent edges are marked for re-examination
  $\Longrightarrow$ Propagation of swapping



Smoothing

- Implemented as optimisation-based vertex smoothing: a vertex u_i is relocated to a new position so that the quality of the worst element among $\{e_{i,0}..e_{i,5}\}$ is maximised (Freitag et al. 1995)
- Linear search problem in the direction of the steepest ascent of the derivative of the quality functional
- Smoothing is propagated

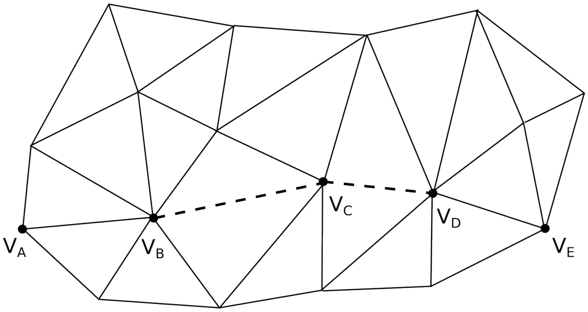


Topological hazards

Example:

- One thread coarsens edge $V_B V_C$, V_B collapses onto V_C
- Another thread coarsens edge $V_C V_D$, V_C collapses onto V_D

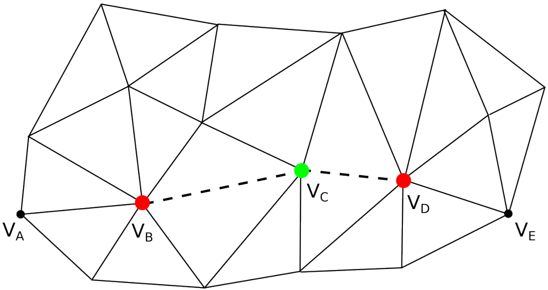
$\Longrightarrow$ V_B collapses onto a vertex (V_C) which is being deleted!



Topological hazards: Mesh colouring

Solution: Mesh colouring

- Nodes are processed in batches of independent sets
  - Guarantees that adjacent nodes cannot collapse at the same time
- colouring is in the loop
  - Need it to be fast and use as few colours as possible
- colouring algorithm by Çatalyürek et al.
  - Based on optimistic/speculative execution
  - We developed an improved version (more on that offline)

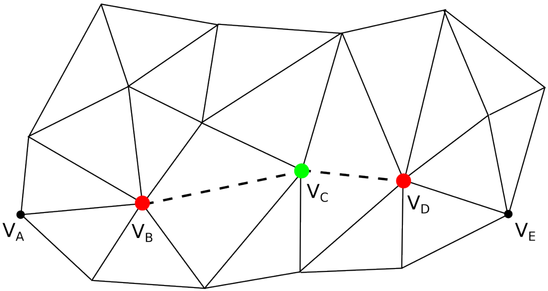


Race conditions

Example: updating adjacency lists

- One thread coarsens edge $V_B V_C$, V_B collapses onto V_C
  - adjacency lists of V_C are modified
  - e.g. V_A must be added to the node-node list of V_C
- Another thread coarsens edge $V_D V_C$, V_D collapses onto V_C
  - adjacency lists of V_C are modified
  - e.g. V_E must be added to the node-node list of V_C

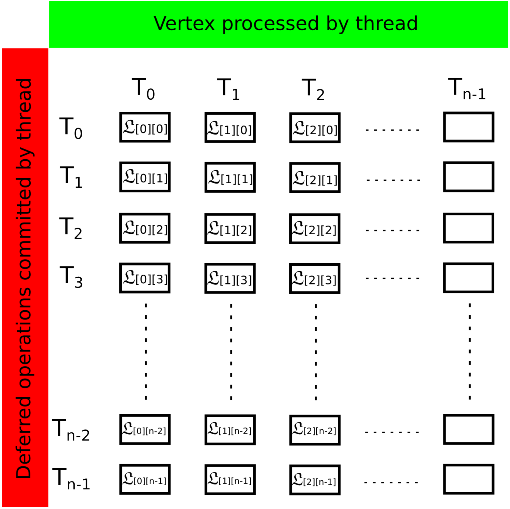
$\Longrightarrow$ Both threads try to modify the node-node list of V_C



Race conditions: Deferred updates

Solution: Defer updates until the independent set has been processed

- Allocate lists $\mathcal{L}_{[i][j]}$ of deferred updates, $i, j = 0..nthreads - 1$
- A thread T_i stores updates pertaining to vertex V_A in $\mathcal{L}_{[T_i][j]}$, $j = hash(V_A)\%nthreads$
- At the end, every thread T_j commits all updates in $\mathcal{L}_{[i][T_j]}$, $i{=}0..N{-}1$
- Advantage: Every thread visits only those updates it is responsible for committing $\Longrightarrow$ FAST!



Worklists

Worklist: A set of workitems which will be processed, e.g. a global worklist of nodes in an independent set

- Threads colour the mesh in parallel
  - Every thread stores the nodes it has coloured in local (private) arrays, $local_{[T_i][colour],\ colour=0..ncolours}$
  - For each colour C, we need to concatenate all private arrays $local_{[T_i][C],\ i=0..N-1}$ into a global array $global_{[C]}$
- Classic approach: Prefix sum (or "scan" in MPI terminology) on the index in $global_{[C]}$ for every thread
  - Threads need to synchronise $\Longrightarrow$ SLOW!
- Alternative: Atomic fetch-and-add
  - Introduced in OpenMP 3.1
  - "atomic capture" directive
  - Older compilers support it either via intrinsics or inline assembly

Worklists: Example

```
// Pre-allocate enough space
std::vector<Item> globalWorkist(some_appropriate_size);
int worklistSize = 0;

#pragma omp parallel
{
  // Initialise a private list
  std::vector<Item> private_list;

#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 global worklist
  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 omp-for
  - Threads need not synchronise at the end of the loop $\Longrightarrow$ FAST!

Loop scheduling: OMP

- Highly diverse loops.
- Example: Mesh refinement
  - Element-refinement loop traverses all elements
  - An element can be processed in 4 different ways: no split, 1:2, 1:3, 1:4

  $\Longrightarrow$ Load imbalance!
- OMP dynamic scheduling
  - Perfect load balance
  - Way too much overhead (millions of nodes/elements)

  $\Longrightarrow$ Poor performance
- OMP guided scheduling
  - Decent load balance, but it could be better
  - Almost no overhead

  $\Longrightarrow$ Much better performance

Loop scheduling: Work-stealing

- Work-stealing scheduler
  - Very good load balance
  - Relatively little overhead

$\Longrightarrow$ Work-stealing is the way to go!

- OMP does not support work-stealing:
  - We had to implement it manually
- Hand-written scheduler implements an improved version of the classic work-stealing algorithm:
  - Excellent load balance
  - Very little overhead

$\Longrightarrow$ Best performance

- Work on this scheduler is still in progress:
  - Preliminary results from synthetic benchmarks: outperforms Intel® Cilk™ Plus work-stealing

PRAgMaTIc

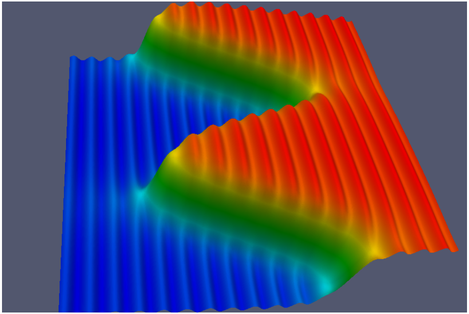
- Parallel anisotRopic Adaptive Mesh ToolkIt:
  - 2D/3D mesh adaptivity framework
  - Open source, under the BSD license
  - Available on Github
    `https://github.com/meshadaptation/pragmatic`
- Implements all aforementioned adaptive algorithms
- Hybrid OpenMP/MPI support
- Currently being integrated with Dolfin (FEniCS) and DMPlex (PETSc)

PRAgMaTIc: Sample benchmark

A synthetic solution ψ is defined to vary in time and space for some value of the period T:

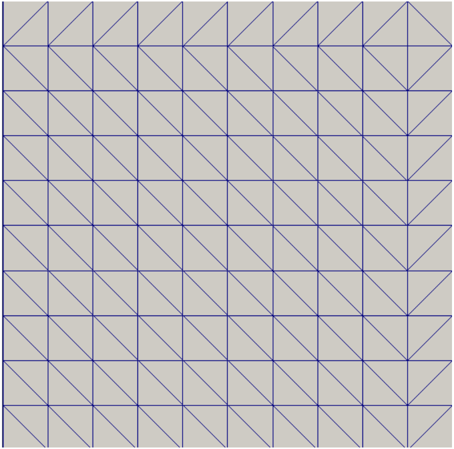
$$\psi(x,y,t) = 0.1\sin\left(50x + \frac{2\pi t}{T}\right) + \arctan\left(-\frac{0.1}{2x - \sin\left(5y + \frac{2\pi t}{T}\right)}\right)$$

Benchmark solution field for some time step t_i



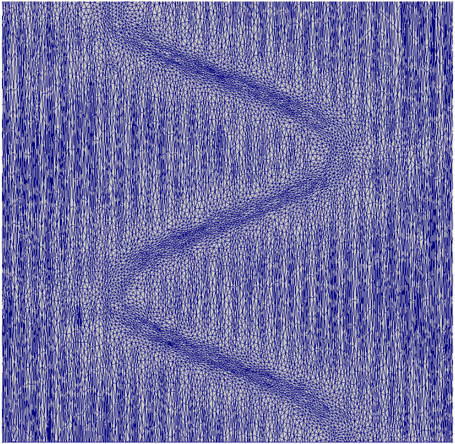
Sample benchmark: Initial mesh

Initial, auto-generated mesh



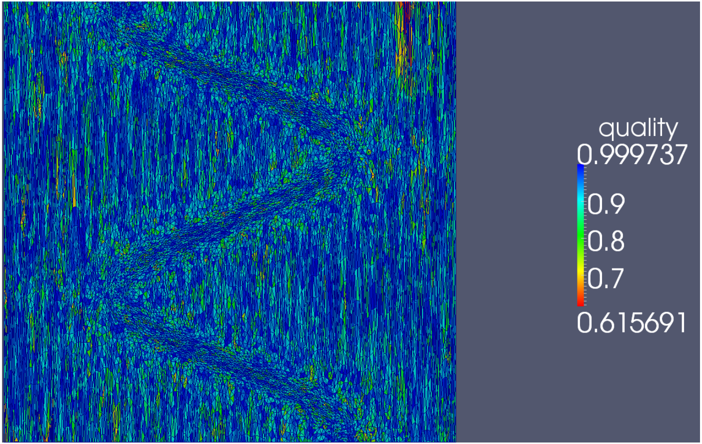
Sample benchmark: Adapted mesh snapshot

Adapted mesh for time step t_i



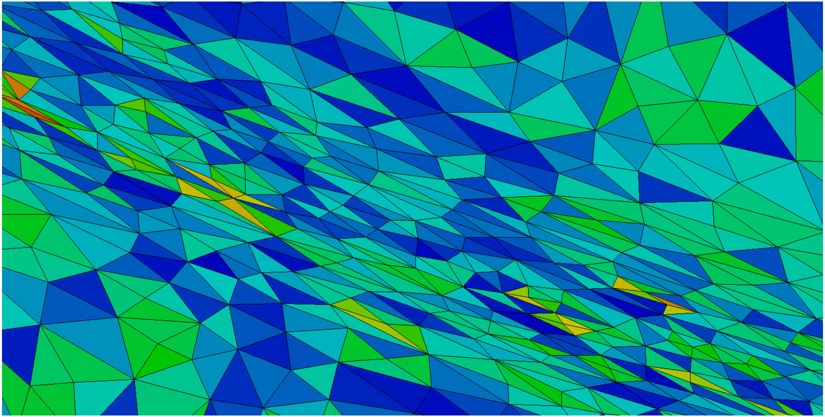
Sample benchmark: Mesh quality snapshot

Quality of adapted mesh for time step t_i



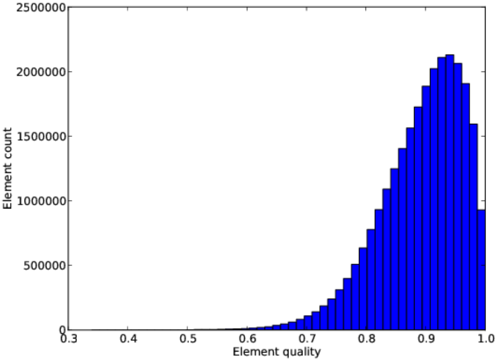
Sample benchmark: Mesh quality detail

Detail of quality around the sinusoidal front



Sample benchmark: Aggregated mesh quality

Aggregated histogram of element quality over all time steps



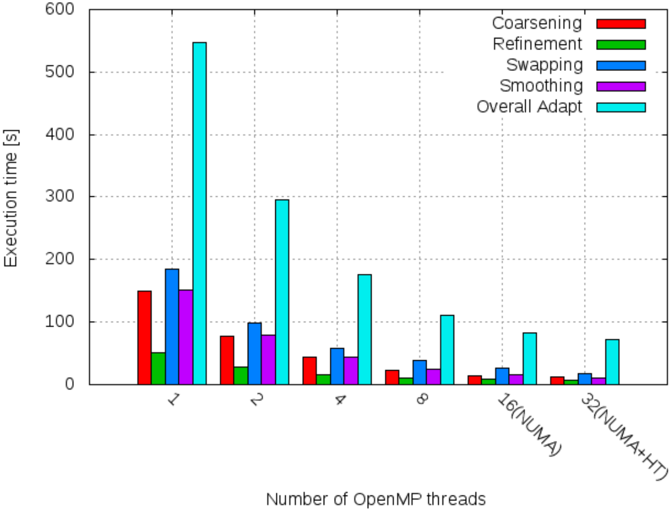
- Average element quality: > 0.9 (close to ideal 1.0)
- Worst element quality: > 0.6

Performance results

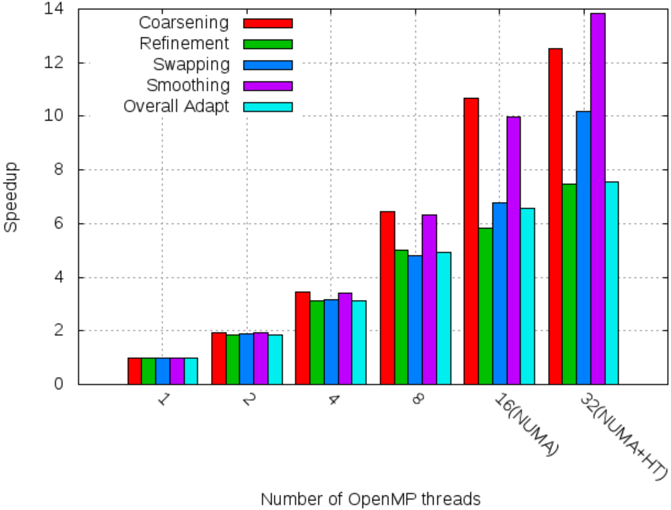
Same sample benchmark

- $x100$ finer metric tensor field, $\approx$ 500k elements, $\approx$ 250k nodes
- Compiled with Intel®Compiler Suite 14.0.1, `-Ofast` flag
- Executed on a dual-socket Xeon®E5-2650 system (Sandy Bridge, 2GHz, 8 cores/16 HT per socket), using thread-core affinity support
- Execution time over all time steps for:
  (1) each of the four adaptive algorithms
  (2) total adapt = sum of the four adaptive algorithms + mesh defragmentation
- $\approx$ 1.5s per time step with 32 threads
- low compared with typical solution times

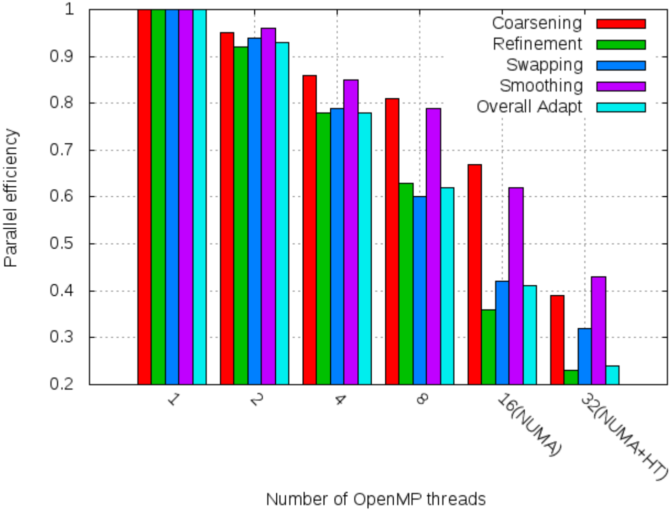
Performance results: execution time



Performance results: speedup



Performance results: parallel efficiency



Performance Results

- Coarsening and smoothing scale well
  - Scalability is mostly limited by thread synchronisation at the end of every independent set
- Refinement and swapping are further affected by bandwidth saturation
  - Enabling hyperthreading improves performance considerably
  - Bandwidth saturation is only to be expected for an application with little data locality

Can we do better?

- Thread synchronisation is the main factor limiting parallel scalability
- colouring and the deferred-operations mechanism involve thread synchronisation
- Alternative: optimistic execution
  - Inspired by the Galois framework (Pingali *et al.*)
  - Lock associated with every mesh vertex
  - A thread tries to acquire the locks of all vertices in a local mesh patch
  - If one of the locks is already held by another thread, abort
  - Early experimentation: abort ratio $< 0.01\%$
  - Single-threaded execution is slower (acquiring/releasing locks is expensive)
  - But code becomes more scalable (Pingali reports parallel efficiency of $> 70\%$ on a 512-core SGI Ultraviolet system)

Conclusions

- PRAgMaTIc produces high-quality adapted meshes
- Anisotropic mesh adaptivity sounds expensive and hard to parallelise
- It can be fast enough to pay off in common usage scenarios
- Some remaining thread synchronisation and bandwidth saturation are currently the limiting factors
- Current focus is on performance optimisation for 3D and MPI
- Inherent difficulty of parallelising complex, irregular algorithms:
  - Optimistic colouring, deferred operations, worklists, work-stealing scheduler proved to be keys to high performance
  - This irregular compute methodology can be used in other applications with mutable irregular data

Acknowledgements and further reading

PRAgMaTIc is brought to you by (alphabetically):

- **Dr. Gerard J. Gorman**, `g.gorman@imperial.ac.uk`, Department of Earth Science and Engineering, Imperial College London, UK
- **Prof. Paul H. J. Kelly**, `p.kelly@imperial.ac.uk`, Department of Computing, Imperial College London, UK
- **Georgios Rokos**, `georgios.rokos09@imperial.ac.uk`, Department of Computing, Imperial College London, UK

This project has been supported by Fujitsu Laboratories of Europe Ltd and the EPSRC (grant numbers EP/I00677X/1 and EP/L000407/1).