Imperial College London
Department of Computing

Scalable Multithreaded Algorithms for Mutable Irregular Data with Application to Anisotropic Mesh Adaptivity

Georgios Rokos

Supervised by Prof. Paul H. J. Kelly
and Dr. Gerard J. Gorman

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This report represents my own work and to the best of my knowledge it contains no materials previously published or written by another person for the award of any degree or diploma at any educational institution, except where due acknowledgement is made in the report. Any contribution made to this research by others is explicitly acknowledged in the report. I also declare that the intellectual content of this report is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Georgios Rokos

Copyright

Abstract

Anisotropic mesh adaptation is a powerful way to directly minimise the computational cost of mesh based simulation. It is particularly important for multi-scale problems where the required number of floating-point operations can be reduced by orders of magnitude relative to more traditional static mesh approaches. Increasingly, finite element/volume codes are being optimised for modern multicore architectures. Inter-node parallelism for mesh adaptivity has been successfully implemented by a number of groups using domain decomposition methods. However, thread-level parallelism using programming models such as OpenMP is significantly more challenging because the underlying data structures are extensively modified during mesh adaptation and a greater degree of parallelism must be realised while keeping the code race-free.

In this thesis we describe a new thread-parallel implementation of four anisotropic mesh adaptation algorithms, namely edge coarsening, element refinement, edge swapping and vertex smoothing. For each of the mesh optimisation phases we describe how safe parallel execution is guaranteed by processing workitems in batches of independent sets and using a deferred-operations strategy to update the mesh data structures in parallel without data contention. Scalable execution is further assisted by creating worklists using atomic operations, which provides a synchronisation-free alternative to reduction-based worklist algorithms. Additionally, we compare graph colouring methods for the creation of independent sets and present an improved version which can run up to 50% faster than existing techniques. Finally, we describe some early work on an interrupt-driven work-sharing for-loop scheduler which is shown to perform better than existing work-stealing schedulers.

Combining all aforementioned novel techniques, which are generally applicable to other irregular problems, we show that despite the complex nature of mesh adaptation and inherent load imbalances, we achive a parallel effi-

ciency of 60% on an 8-core Intel$^{®}$Xeon$^{®}$ Sandy Bridge and 40% using 16 cores on a dual-socket Intel$^{®}$Xeon$^{®}$ Sandy Bridge ccNUMA system.

Acknowledgements

I would like to express my gratefulness and appreciation to the following people for their contribution to the accomplishment of this thesis:

- Prof. Paul H.J. Kelly, my first supervisor, who has guided and shaped my work since I first arrived at Imperial College London. His enthusiasm, patience and dedication have been crucial for fulfilling the requirements of my PhD degree.
- Dr. Gerard J. Gorman who supported me throughout this project, giving me invaluable advice and guidance, providing all necessary algorithmic knowledge and assuming the role of the second supervisor, devoting a great amount of his time.
- Dr. Carlo Bertolli for interesting discussions and advice on various general topics and whose company I have enjoyed very much.
- Anastasia Eleftheriou whose company and support at some stages of my life as a PhD student has been second to none.
- My family who provided emotional support against all difficulties of living away from my home country.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Thesis Statement

This thesis argues that algorithms for irregular data with mutable dependencies can be parallelised efficiently on shared-memory systems. By deferring data write-back to specific points in the execution, carefully using atomic operations on parallel worklists and executing parts of an algorithm speculatively, it is possible to reduce the amount of thread synchronisation and make the parallel algorithm scalable, retaining high levels of parallel efficiency on multicore and manycore platforms.

1.2 Motivation and Objectives

Finite element (FEM) and finite volume methods (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. In addition, simplices are well suited to smoothinly adjusting the resolution of the mesh throughout the domain, allowing for local refinement of the mesh without hanging nodes.

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 [90, 105]. Since many of these applications include a strong requirement to conform to complex geometries or to resolve a multiscale 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.

A difficulty with mesh based modelling is that the mesh is generated before the solution is known, however, the local error in the solution is related to the local mesh resolution. Resolution in time and space are often the limiting factors in achieving accurate simulations for real world problems across a wide range of applications in science and engineering. The brute force strategy is typical, whereby a user varies the resolution at the mesh generation phase and reruns the simulation several times until the required accuracy is achieved. This is successful up to a point when a numerical method is relatively straightforward to scale up on parallel computers, for example finite difference or lattice Boltzmann methods. However, this approach is inefficient, often lacks rigour and may be completely impractical for multiscale time-dependent problems where superfluous computation may well be the dominant cost of the simulation. In practice, this means simulation accuracy is usually determined by the available computational resources and an acceptable time to solution rather than the actual needs of the problem.

Anisotropic mesh adaptation methods provide an important means to minimise superfluous computation associated with over resolving the solution while still achieving the required accuracy, *e.g.* [83, 93, 17, 4, 89, 71]. 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.

Parallel computing - in order to make use of larger compute resources - provides an obvious source of further improvements in accuracy. Previous work has described how adaptive mesh methods can be implemented in parallel in the context of distributed memory parallel computers using MPI ([71, 40, 28, 6]). Therefore, both adaptive mesh methods and parallel computing can be combined to achieve scalable and efficient high fidelity simulations. However, this comes at the cost of further overheads, including the need to manage 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 and reduced amount of memory per core in the world's most powerful supercomputers. For example, each node of Fujitsu's "K computer" consists of an 8-core SPARC64™ VIIIfx CPU [44, 114] and the SPARC64™ IXfx-based nodes in Fujitsu's PRIMEHPC FX10 machines have 16 cores per CPU [43]. Similarly, Cray XE6™ "Blue Waters" nodes are made up of two 12-core AMD Opteron™ 6100 processors (24 cores per node) [57, 60], IBM®'s Blue Gene®/Q nodes each have 16 cores for computation [53], Intel®'s latest Haswell-based CPUs contain up to 18 cores [26] and its MIC co-processors have over 60 cores [59, 97, 27]. It is assumed that the nodes of a future exascale supercomputer will each contain thousands or even tens of thousands of cores [33].

For this reason it is important that algorithms are developed with very high levels of parallelism. On such architectures, a popular parallel programming paradigm is to use a thread-based parallel API, such as OpenMP [29], to exploit shared memory within a shared memory node and a message passing API such as MPI [110], for interprocess communication. When the computational intensity is sufficiently high, a third level of parallelisation may be implemented via SIMD instructions, such as SSE or AVX, at the core level.

OpenMP itself is evolving to keep pace with these trends. Version 3.0 [86] moved beyond parallel loops and introduced the concept of generalised tasks with complex and dynamic control flows to support irregular parallelism. In version 3.1 [87] the OpenMP Architecture Review Board extended atomics to support `capture` and `write` operations, added `min` and `max` reduction operators, refined the tasking model with `final` and `mergeable` clauses and provided initial support for thread binding. In its latest version, 4.0 [88], the OpenMP standard has adopted support for user-defined reductions, accelerator offloading, SIMD constructs, stronger thread-core affinity and sequentially consistent atomics. Plans for future versions include support for memory affinity, transactional memory and thread-level speculation, additional synchronisation mechanisms, locality optimisations and runtime decisions about scheduling to support dynamic resource allocation and load balancing [34, 16].

Many algorithms are inherently sequential; they fall into the class of P-complete problems. Examples include the Circuit Value Problem, Lexicographically First Depth-First Search Ordering and Context-Free Grammar Membership. Other algorithms are hard to parallelise effectively, *e.g.* the Greatest Common Divisor of two numbers. A thorough analysis of such algorithms can be found in [50]. For other classes of algorithms, however, while it can be difficult to achieve a sufficiently high level of parallelism at the algorithm level, there are many opportunities to improve performance and scalability by reducing communication needs, memory consumption, resource sharing, improved load balance and other algorithmic changes [96].

For the class of graph algorithms, which is the main topic of this thesis, there are many factors which limit performance and scalability [76]:

- The irregular nature of graphs leads to unpredictable memory access patterns, rendering prefetching techniques inapplicable. This makes memory accesses costly.
- Another result of graphs' irregularity is poor data locality. Modern architectures are equipped with fast caches and rely on spatial and temporal data locality to achieve high performance. Therefore, even a serial graph algorithm can perform poorly due to little data reuse.
- Even worse, graph kernels are mainly memory-bound rather than compute-bound and so the execution is dominated by data access latency. In fact, there are cases where there is no computation on the data at all, *e.g.* many graph colouring algorithms, therefore nothing can be done to hide that latency.
- Extracting parallelism can be hard, since many algorithms are data-driven, meaning that the exact operations to be performed on a vertex are determined by that particular vertex and are not known a priori. This can easily lead to imbalances in workload when using static partitioning or coarse-grained parallelism.
- When parallelism is fine-grained, threads need to synchronise frequently, so sequential parts appear in between parallel regions inside a procedure.

For the aforementioned reasons, adaptive mesh algorithms sound hard to parallelise effectively on modern shared-memory architectures. In this thesis we take a fresh look at anisotropic adaptive mesh methods (also known as mesh optimisation methods) in 2D and describe new scalable parallel techniques suitable for modern multicore and manycore architectures. This work builds upon: the adaptive procedure described by [71] which uses a combination of coarsening, refinement and swapping to adapt the mesh; the optimisation-based vertex smoothing algorithm by [37] to fine-tune element quality; the general parallel framework proposed by [40] which ensures thread-safe execution.

1.3 Application

The work presented in this thesis forms the basis of the open source code PRAgMaTIc[1] (Parallel anisotRopic Adaptive Mesh ToolkIt), which has been integrated into the open source computational fluid dynamics software Fluidity[2].

1.4 Contributions

In this research we examine the anisotropic adaptive mesh methods in 2D as a case study to develop new scalable thread-parallel algorithms suitable for modern multicore 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 key contributions are:

- We present scalable parallel techniques which form an algorithmic framework for problems with mutable irregular data. These techniques include (a) vertex colouring and independent sets to extract parallelism, (b) design choices regarding the representation of irregular data which lead to as few data structures as possible, (c) the deferred updates strategy, according to which updates to shared data structures are committed at selected points throughout the execution of an algorithm in order to avoid data contention and races and (d) handling of parallel worklists with the assistance of atomic-capture operations. This irregular compute methodology is described in Chapter 4.

[1]https://github.com/ggorman/pragmatic
[2]http://fluidityproject.github.io/

- We discuss previous work on graph colouring and demonstrate an improved parallel greedy colouring algorithm for shared-memory environments which outperforms its predecessors. There are two sources of speedup: (a) reduced number of thread barriers and (b) higher thread divergence throughout the execution. This case provides evidence that thread divergence in speculative parallel execution can contribute toward minimising the need for rolling-back and, as a consequence, the algorithm runs to completion in less time. This work is covered in Chapter 5.

- We show how the aforementioned parallel techniques are applied to adaptive mesh algorithms, leading to the creation of PRAgMaTIc, the first (to the best of our knowledge) threaded implementation of anisotropic mesh adaptation. We present a systematic performance evaluation that (a) shows the potential of a parallel efficiency of 60% on an 8-core UMA architecture and 40% on a 16-core ccNUMA architecture and (b) characterises what the performance depends on and where the bottlenecks are. This is the focus of Chapter 6.

- We present our early work on an interrupt-driven work-sharing scheduler (IDWS) for OpenMP which is shown to be a better all-around option compared with current built-in for-loop schedulers. IDWS performs better than classic work stealing thanks to two key features not found in other schedulers: (a) idle threads use a heuristic method for finding the most loaded worker to request work from and (b) the request is sent using hardware interrupts so as to get a response from the worker as promptly as possible. IDWS is described in Chapter 7.

1.5 Dissemination

The work contained within this thesis has been disseminated to a wider community through publications, presentations, and the release of software under open-source licences. The list of publications is as follows:

[101] **Georgios Rokos**, Gerard Gorman, and Paul H.J. Kelly. **Accelerating anisotropic mesh adaptivity on nvidias cuda using texture interpolation.** In Emmanuel Jeannot, Raymond Namyst,

and Jean Roman, editors, Euro-Par 2011 Parallel Processing, volume 6853 of Lecture Notes in Computer Science, pages 387398. Springer Berlin Heidelberg, 2011. This paper presents my seminal work on mesh adaptivity by studying the Laplacian vertex smoothing algorithm and parallelising it for CUDA.

[49] GJ Gorman, J Southern, PE Farrell, MD Piggott, **G Rokos**, and PHJ Kelly. **Hybrid OpenMP/MPI anisotropic mesh smoothing.** Procedia Computer Science, 9:15131522, 2012. This paper is the continuation of [101], giving insight into the first algorithm implemented in PRAgMaTIc, a hybrid OpenMP/MPI version of vertex smoothing. The OpenMP part was implemented by my co-authors, based on my CUDA code from the previous publication.

[100] **Georgios Rokos** and Gerard Gorman. **PRAgMaTIc - Parallel Anisotropic Adaptive Mesh Toolkit.** In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, Facing the Multicore Challenge III, volume 7686 of Lecture Notes in Computer Science, pages 143144. Springer Berlin Heidelberg, 2013. Abstract and poster describing my early work on all four adaptive algorithms.

[48] Gerard J. Gorman, **Georgios Rokos**, James Southern, and Paul H. J. Kelly. **Thread-parallel anisotropic mesh adaptation.** Accepted for publication in Proceedings of the 4th Tetrahedron Workshop on Grid Generation for Numerical Computations, 2014. This paper encompasses my work described in Chapters 4 and 6, essentially presenting the final, optimised and scalable version of the 2D branch of PRAgMaTIc.

Progress on PRAgMaTIc has been presented on the following occasions:

- Poster at the *CARE (Computation for Advanced Reactor Engineering)* meeting, 23 January 2014.
- Talk at *Recent Advances in Parallel Meshing Algorithms minisymposium, SIAM Conference on Parallel Processing and Scientific Computing, 18-21 February 2014, Portland, OR, USA.*

Papers on the work presented in Chapters 5 and 7 are currently under preparation.

1.6 Thesis Outline

The rest of this report is laid out as follows: Chapter 2 covers background theory behind anisotropic PDEs and the optimisation algorithms which control solution error and improve mesh element quality. Chapter 3 surveys related work in parallel mesh adaptivity and similar morph algorithms. Because this multidisciplinary work covers a number of specialist areas, specific literature review on other topics is presented in the relevant chapters. In Chapter 4 we explore design choices and generic scalable parallel techniques which aid in the parallelisation of mesh adaptivity kernels. Chapter 5 reviews previous work on parallel graph colouring methods, describes an improved algorithm and shows how speculative execution can improve multithreaded performance. Chapter 6 demonstrates how the techniques from Chapters 4 and 5 are combined with the adaptive kernels from Chapter 2 to create a scalable shared-memory mesh adaptivity framework. In Chapter 7 we propose an interrupt-driven for-loop scheduler and present some preliminary results from using it both in synthetic benchmarks and in adaptivity kernels. Finally, we conclude the thesis in Chapter 8.

2 Background Theory

In this chapter 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 PDEs and the Finite Element Method

The Finite Element Method (FEM) is a common numerical approach for the solution of PDEs, in which the problem space is discretised into smaller elements, usually of triangular (in 2D) or tetrahedral (in 3D) shape. Many kinds of elements can be used, *e.g.* quadrilaterals and hexes, but we are focusing on simplices as there are robust mesh generation methods for complex geometries using simplices and also mesh adaptivity has been studied extensively on simplicial meshes. The Finite Element Method uses parts of the work presented in this thesis but covering it is out of scope. An excellent reference for finite element analysis can be found in [116].

Mesh quality impacts discretisation error. A low quality mesh affects both convergence speed and solution accuracy [37]. Error estimates on the PDE solution help evaluate a quality functional [111] and determine the low-quality elements, which a mesh-improving algorithm tries to adapt towards the correct solution.

Finite element and finite volume methods on unstructured meshes offer significant advantages for many real world applications. For example, meshes comprised of simplices that conform to complex geometrical boundaries can now be generated with relative ease. In addition, simplices are well suited to varying the resolution of the mesh throughout the domain, allowing for local coarsening and refinement of the mesh without hanging nodes. It is common for these codes to be memory bound because of the indirect addressing and the subsequent irregular memory access patterns that the unstructured data structures introduce [93]. However, discontinuous Galerkin and high-order finite element methods are becoming increasingly popular because their numerical properties and associated compact data structures allow data to be easily streamed on multi-core architectures.

2.2 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.

A solution to this is to compute appropriate local error estimates *a posteriori* and use this information to compute a field on the mesh which specifies the local mesh resolution requirement. In the most general case the desired resolution is specified as a metric tensor field where the eigenvalues encode the local resolution in the direction of the associated eigenvector. Use of a metric tensor field consequently accommodates for anisotropic problems, i.e. resolution requirements can be specified anisotropically; for a review of the procedure see [41]. Size gradation control can be applied to this metric tensor field to ensure that there are not abrupt changes in element size [5].

A posteriori error estimation either for computational error control using goal-oriented functionals or classic error control in global energy norms is a field in which research is taking place very actively. This topic goes beyond the scope of this thesis; the reader is referred to the literature (*e.g.* [104, 9, 67] and the publications cited therein) for an extensive coverage of error control.

2.3 Anisotropic Problems

In many applications the resolution requirement is anisotropic; *e.g.* higher resolution is required perpendicular to a shock front than along the shock.

A problem is characterised as anisotropic if its solution exhibits directional dependencies, i.e. an anisotropic mesh contains elements which have some (suitable) orientation.

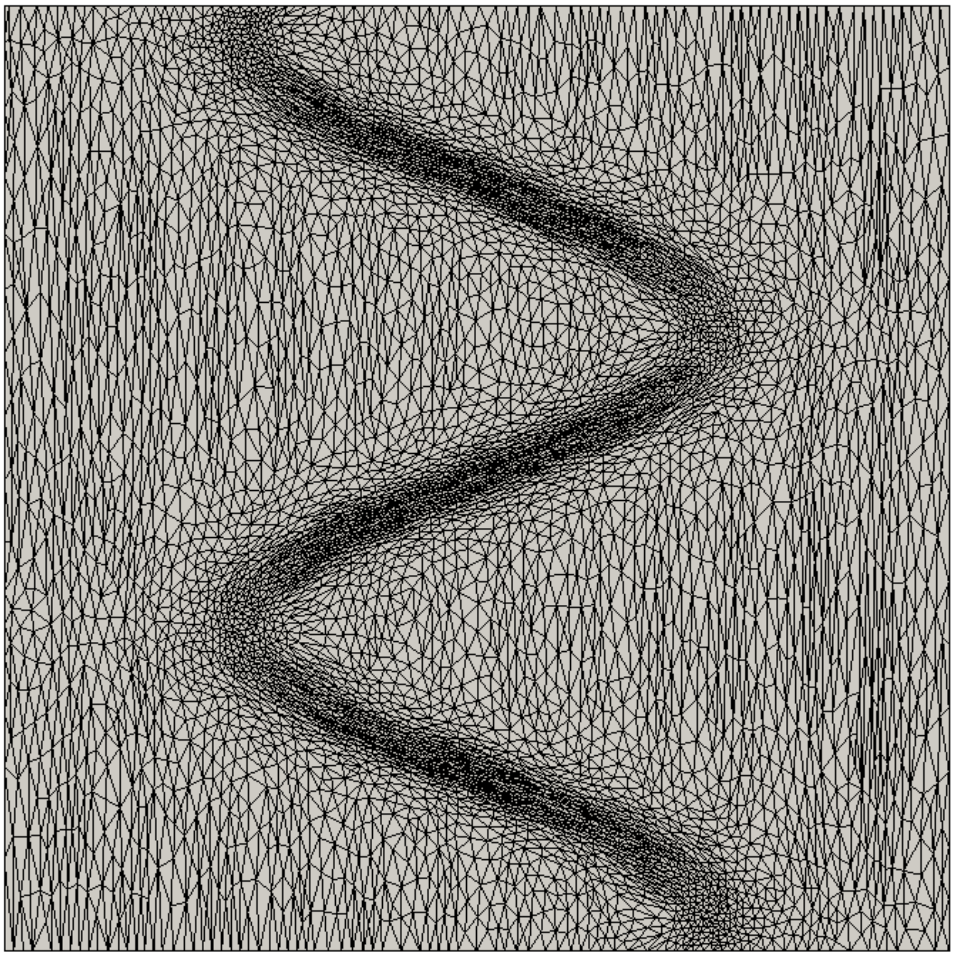


Figure 2.1: Example of an anisotropic mesh, in which higher resolution in required along a sinusoidal front.

An example of an anisotropic mesh is shown in Figure 2.1, where higher resolution in required along a sinusoidal front. A magnified depiction of the central region can be seen in Figure 2.2, which exposes in higher detail mesh elements in the highly anisotropic area around the front. Those triangles are stretched along the front direction. Different cases of space distortion are found in other regions of the mesh, *e.g.* the top left corner where elements

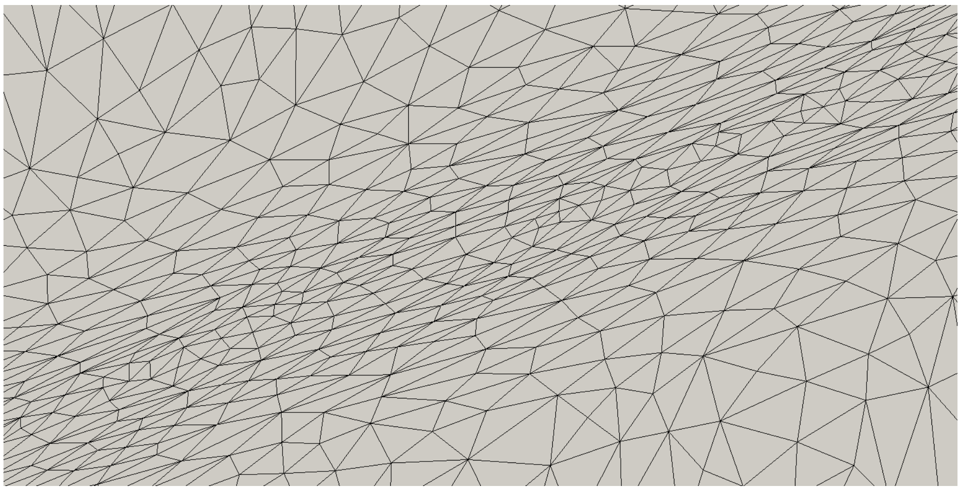


Figure 2.2: Magnification of the centre of mesh from Figure 2.1 around the sinusoidal front, exposing in finer detail some mesh elements in the highly anisotropic area. Elements are stretched along the direction of the front. Moving away from the front, the mesh becomes more uniform.

are orientated along the y-axis, as is shown in the magnification in Figure 2.3. Another example of gradually adapting a 3D mesh to the requirements of an anisotropic problem can be seen in Figure 2.4 [89].

The error estimation gives information about how big or small a mesh element should be. In 1-D, the solution error inside an element e (i.e. a line segment) is defined as

$$\varepsilon = h_e^2 \mid \frac{\partial^2 \psi}{\partial x^2} \mid, \tag{2.1}$$

where h_e is the length of element e and ψ is the solution variable. In multi-dimensional problems, the error is defined as

$$\varepsilon = \mathbf{u}^T \mid \mathbf{H} \mid \mathbf{u}, \tag{2.2}$$

where $\mathbf{H}$ is the Hessian of the solution equation and $\mathbf{u}$ is a vector which shows the ideal length and orientation of element e [115]. Simply, the higher the error inside an element the smaller this element has to become [89].

In the last equation, the vector $\mathbf{u}$ is constructed according to a metric tensor $\mathbf{M}$, i.e. a tensor which, for each point in the 2D (or 3D) space,

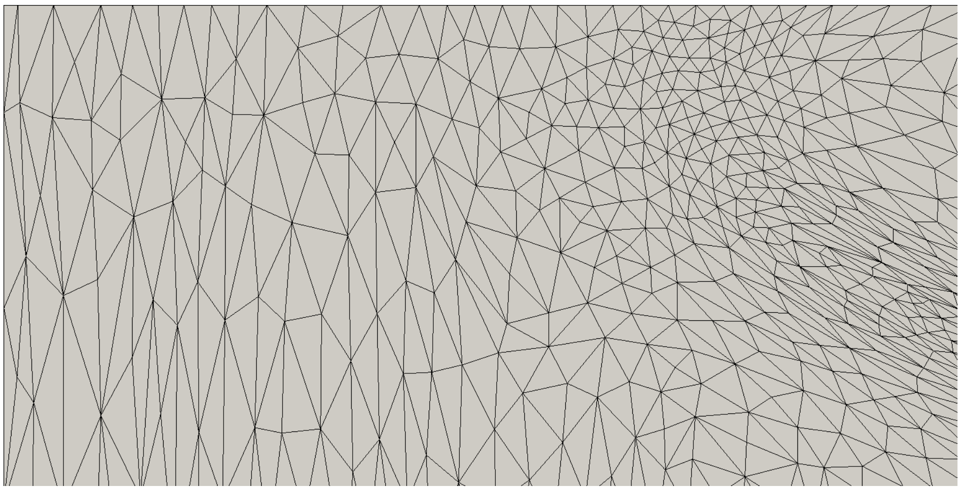


Figure 2.3: Magnification of the top left corner of mesh from Figure 2.1 showing that most elements are stretched along the y-axis.

represents the desired length and orientation of an edge containing this point. The metric tensor is discretised node-wise. The value of the metric along a mesh edge can be taken by linearly interpolating the metric from the edge vertices.

A metric tensor is essentially a symmetric matrix which defines length of vectors and allows us to calculate inner products in generalised spaces in the same way the dot product is used to define distance in the standard Euclidean space. As an example in 2D, let an edge be defined by vertices $V_1(x_1, y_1)$ and $V_2(x_2, y_2)$; then the edge is represented by vector $\mathbf{E} = (x_0, y_0) = (x_2 - x_1, y_2 - y_1)$. The Euclidean length of that edge is given by the dot product:

$$L_{Euclidean} = \parallel \mathbf{E} \parallel = \sqrt{\mathbf{E} \cdot \mathbf{E}} = \sqrt{x_0^2 + y_0^2} \tag{2.3}$$

Analogously, the edge's length with respect to a metric tensor $\mathbf{M} = \begin{bmatrix} A & B \\ B & C \end{bmatrix}$

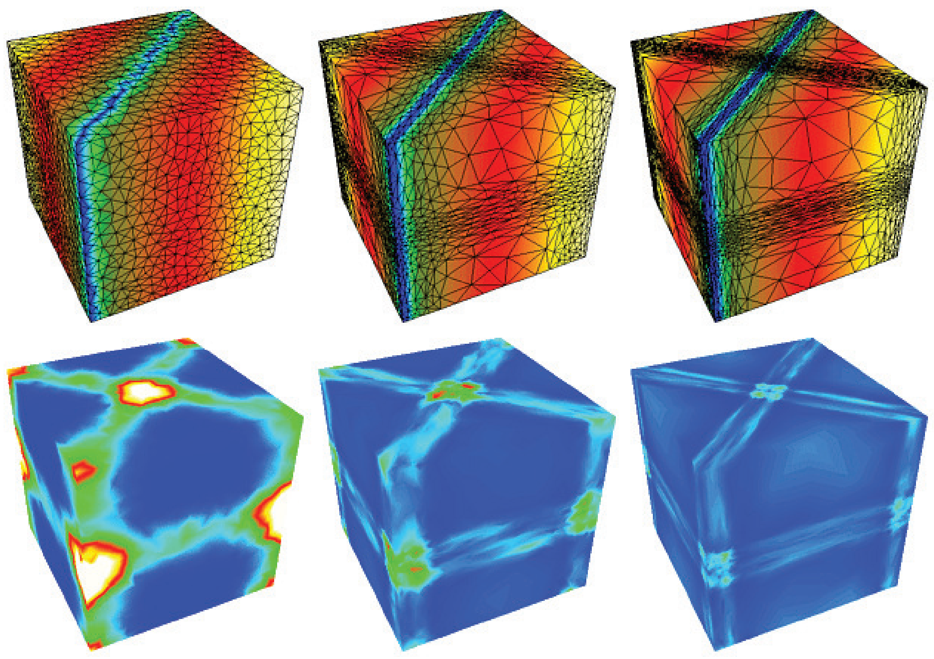


Figure 2.4: Example of anisotropic mesh adaptation. On the top left, solving the PDE on an automatically triangulated mesh results in high solution errors. Error estimations, as depicted in the bottom left figure, indicate areas in which mesh resolution should be focused. After one adaptation step (middle figures) mesh quality has been improved and solution error is greatly reduced. After two iterations (right figures) the results are even better. (figure from [89])

(the tensor is symmetric) can be calculated using the inner product:

$$\begin{aligned} L_M = \parallel \mathbf{E} \parallel_M = \sqrt{\mathbf{E}^T \mathbf{M} \mathbf{E}} = \sqrt{\begin{bmatrix} x_0 y_0 \end{bmatrix} \begin{bmatrix} A & B \\ B & C \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}} = \\ = \sqrt{x_0^2 A + 2x_0 y_0 B + y_0^2 C} \end{aligned} \tag{2.4}$$

The metric is defined in such a way that an edge of an element is of unit length with respect to this metric if it has the desired error ϵ_u indicated by this metric, i.e.

$$\mathbf{M} = \frac{1}{\epsilon_u \mid \bar{H} \mid}. \tag{2.5}$$

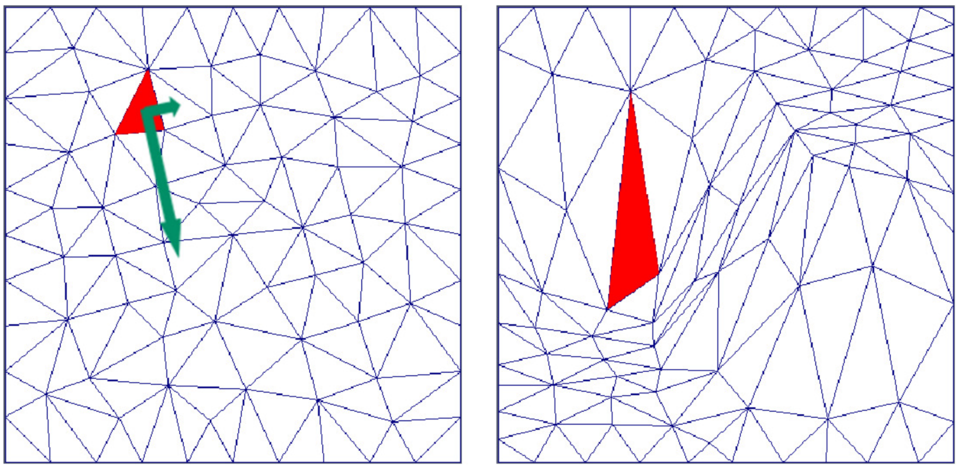


Figure 2.5: Example of a metric tensor in 2D. The green arrows on the left are the eigenvectors of that tensor, each one scaled (multiplied) by the corresponding eigenvalue. Geometrically, the tensor shows the direction to which the red triangle has to be stretched and the amount of distortion required per component of that direction. The result of stretching the triangle is shown on the right.

The metric tensor M can be decomposed as

$$\mathbf{M} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T, \tag{2.6}$$

where $\mathbf{\Lambda}$ is a diagonal matrix the non-zero elements of which are the eigenvalues of $\mathbf{M}$, and $\mathbf{Q}$ is an orthonormal matrix the rows of which are the corresponding eigenvectors $\mathbf{Q}^i$ for each eigenvalue λ^i. Geometrically, $\mathbf{Q}$ represents a rotation of the axis system so that the base vectors show the direction to which the element has to be stretched and $\mathbf{\Lambda}$ represents the amount of distortion (stretching). Each eigenvalue λ^i represents the ideal length of an edge in the direction $\mathbf{Q}^i$. An example is shown in Figure 2.5. If we denote the diagonal values of the Hessian of the solution as h^i, then each eigenvalue λ^i is defined as

$$h^i = \frac{1}{\sqrt{\lambda^i}}, \tag{2.7}$$

so that stretching or compressing an element will be done in an inverse square fashion with respect to the error metric [93].

2.4 Element Quality

Many different measures of element quality have been proposed. An excellent review of different Euclidean geometric metrics for mesh generation applications is given in [64] for 2D and [65] for 3D. However, as mesh generation is part of the preprocessing state, these metrics are not designed to 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 [17, 111, 4, 108, 89].

In the work described here, we use the element quality measure for triangles proposed by Vasilevskii et al. [111]:

$$q^M(\triangle) = \underbrace{12\sqrt{3}\frac{|\triangle|_M}{|\partial\triangle|_M^2}}_{I}\underbrace{F\left(\frac{|\partial\triangle|_M}{3}\right)}_{II}, \tag{2.8}$$

where $|\triangle|_M$ is the area of element $\triangle$ and $|\partial\triangle|_M$ is its perimeter as measured with respect to 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, $|\triangle| = l^2\sqrt{3}/4$ and $|\partial\triangle| = 3l$; and so term $I = 1$. For non-equilateral triangles, $I < 1$. The second factor (II) controls the size of element $\triangle$. The function F is smooth and defined as:

$$F(x) = \left[min\left(x, \frac{1}{x}\right)\left(2 - min\left(x, \frac{1}{x}\right)\right)\right]^3, \tag{2.9}$$

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 in (2.8) 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.

In an anisotropic problem we can use the quantities of area and perimeter if we express them with respect to a non-Euclidean metric $M(\mathbf{x})$. For an element E with area $\mid \Delta \mid_E$ and edges of length $\mathbf{e}_i$ in the standard Euclidean space, its area with respect to the metric $M(\mathbf{x})$ can be calculated as

$$\mid \Delta \mid_M = \sqrt{\det(M)} \mid \Delta \mid_E \tag{2.10}$$

and its perimeter as

$$\mid \partial\Delta \mid_M = \sum_{i=1}^{3} \parallel \partial\mathbf{e}_i \parallel_M = \sum_{i=1}^{3} \sqrt{\mathbf{e}_i^T M \mathbf{e}_i}, \tag{2.11}$$

where we consider that M is constant over the element E.

Adapting a mesh so that it distributes the error uniformly over the whole mesh is, in essence, equivalent to constructing a uniform mesh consisting of equilateral triangles with respect to the metric M. From Figure 2.6 we can see that the ideal element is isotropic in metric space which means that it will look anisotropic (elongated, stretched, aligned to physical solution features) in Euclidean space.

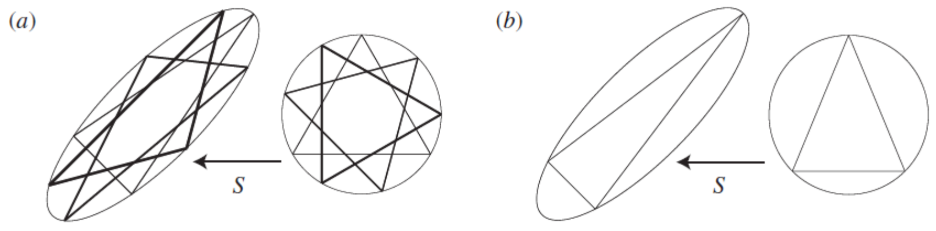


Figure 2.6: Example of mapping of triangles between the standard Euclidean space (left shapes) and metric space (right shapes). In case (α), the elements in the physical space are of the desired size and shape, so they appear as equilateral triangles with edges of unit length in the metric space. In case (β), the triangle does not have the desired geometrical properties, so it does not map to an equilateral triangle in the metric space.(Figure from [93])

2.5 Overall Adaptation Procedure

Mesh improving techniques fall into two main categories, h-adaptivity and r-adaptivity algorithms. The first category contains techniques which try to adapt the mesh by changing its topology. This can be done by removing existing mesh elements via edge collapse (§2.6.1); increasing local mesh resolution via edge refinement by adding new elements, a procedure called refinement (§2.6.2); or replacing a group of elements with a different group via element-edge swaps (§2.6.3); The second group of adaptive algorithms encompasses a variety of vertex smoothing techniques (§2.6.4), all of which leave mesh topology intact and only attempt to improve quality by relocating mesh vertices. While coarsening and refinement control the local resolution, swapping and smoothing are used to improve the element quality.

Algorithm 1 gives a high level view of the anisotropic mesh adaptation procedure as described by [71]. 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. Coarsening is initially applied to reduce the subsequent computational and communication overheads. The second stage involves the iterative application of refinement, coarsening and mesh swapping to optimise the resolution and quality of the mesh. This is called the "h-adaptivity" loop. 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 (e.g. quality-constrained Laplacian smoothing [39], optimisation-based smoothing [37]), which aims primarily at improving worst-element quality. Smoothing is a fairly expensive computational kernel which makes fine changes to the mesh. On the contrary, the other three algorithms are less computationally demanding and make grosser mesh modifications. Including smoothing in the main loop considerably slows down the mesh optimisation procedure for no real benefit in terms of mesh quality, since subsequent h-adaptivity sweeps can change local quality to a great extent. It only makes sense to fine-tune element quality once mesh topology has been fixed.

Algorithm 1 Mesh optimisation procedure.

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

2.6 Adaptation Kernels

2.6.1 Edge Coarsening

Coarsening is the process of lowering mesh resolution locally by removing mesh elements, leading to a reduction in the computational cost. There are two adaptive methods that fall into this category, edge collapse and element collapse. An edge collapses by reducing it into a single vertex. This way, the two elements which share this edge are deleted. An example of this operation is shown in Figure 2.7. Edge collapse is an oriented operation, meaning that an edge can be reduced into a single vertex by following two opposite directions, each one resulting in a different local patch. Element collapse is a similar operation in which an element is reduced into a single vertex, resulting in the deletion of four elements, the element which collapsed plus the three elements which shared an edge with that element. Element collapse is not implemented in PRAgMaTIc.

An algorithm for coarsening has been proposed by Li *et al.* [71]. The algorithm operates on a list of candidate-edges. An edge is marked as being a candidate if its length is shorter than a user-defined minimum length L_{min}. The goal is to remove as many candidate-edges as possible without creating edges longer than a user-defined maximum length L_{max}. Since we are working in metric space, all lengths are calculated with respect to the metric tensor field.

The coarsening kernel is described in Algorithm 2. The process starts by iterating over the list of short edges, inserting all vertices that bound these edges to a dynamic list. After this step, the algorithm loops over the dynamic list, each time choosing an unprocessed vertex from the list. For

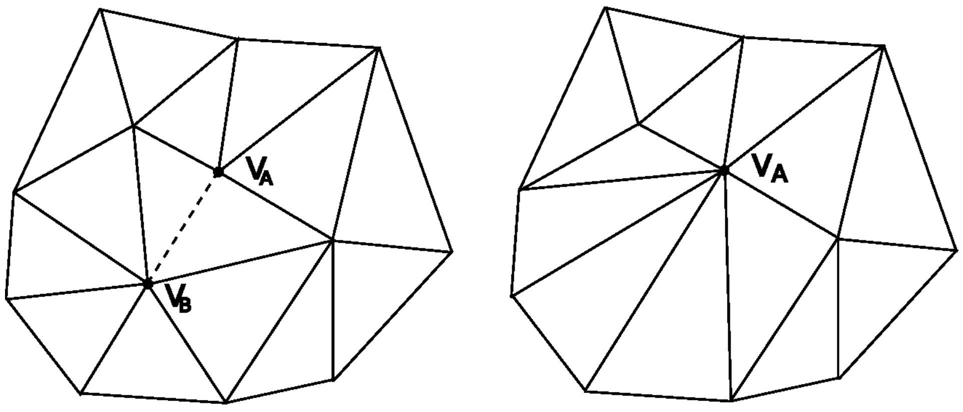


Figure 2.7: Example of edge collapse. The dashed edge in the left figure is reduced into a single vertex by bringing vertex V_B on top of vertex V_A, as can be seen in the middle figure. The two elements that used to share the dashed edge are deleted. Edge collapse is an oriented operation, i.e. eliminating 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.

the chosen vertex V_i, the algorithm decides whether any of the edges connected to it can collapse with the removal of V_i, according to the criteria mentioned above, *i.e.* the length of this edge must be less than L_{min} and after its collapse all other edges connected to V_i must remain shorter than L_{max} and no elements must be inverted as a result of this operation. If the edge can collapse, the algorithm applies the operation, marks all vertices adjacent to V_i as unprocessed and removes V_i from the mesh and the dynamic list. Marking adjacent vertices as unprocessed serves the purpose of propagation. Since the local neighbourhood has been modified by the coarsening operation, those vertices must be (re-)examined for collapse.

As can be seen in Algorithm 2, unprocessed vertices are chosen from the dynamic list in a controlled way, so that they are processed "topologically every other one". The purpose of enforcing this order is to maintain a good vertex distribution and avoid the creation of excessively long edges. As an example, in the local patch in Figure 2.8, if vertex V_B collapses onto V_C and right after that V_C collapses onto V_D, the result will be an excessively long edge $\overline{V_A V_D}$. This processing schedule can be made explicit via colouring. In this context, colouring the mesh is a required preprocessing step not only

Algorithm 2 Serial edge collapse algorithm by Li *et al.* [71].

```
initialize a dynamic vertex list $L_D \leftarrow \emptyset$
loop over the list of short edges
    for each vertex $V_i$ that bounds the edge do
        if $V_i$ is not in $L_D$ then
            append $V_i$ to $L_D$
        end if
    end for
end loop
while there are unprocessed vertices in $L_D$ do
    choose a vertex $V_i$ from $L_D$ using the "every other vertex" rule
    $S_i \leftarrow$ the set of all edges connected to $V_i$
    $E_j \leftarrow$ shortest edge in $S_i$
    if length of $E_j > L_{min}$ then        ▷ no adjacent edge can be removed
        remove $V_i$ from $L_D$
    else
        evaluate collapse of $E_j$ with the removal of $V_i$
        if $(\forall E_i \in S_i : \text{length}(E_i) \leq L_{max})$ and($\nexists$ inverted elements) then
            apply collapse
            mark all vertices adjacent to $V_i$ as unprocessed
            remove $V_i$ from $L_D$
        else
            mark $V_i$ as processed                ▷ and $E_j$ does not collapse
        end if
    end if
end while
```

for the parallel algorithm (as will be discussed in Section 4.1) but also for the serial one.

2.6.2 Element Refinement

Refinement is the process of increasing mesh resolution locally. Although it leads to an increase in the computational cost, refinement is a key process in improving mesh quality.

The term refinement encompasses two operations: splitting of edges and subsequent division of elements. When an edge is longer than desired it is bisected, giving two shorter halves. 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

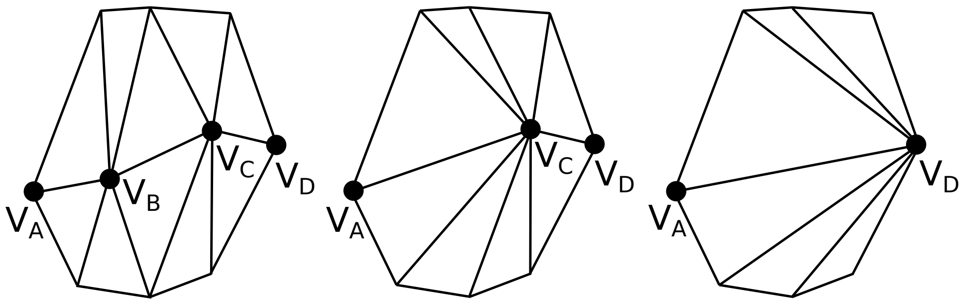


Figure 2.8: Example of edge collapse resulting in excessively long edges. If vertex V_B collapses onto V_C and right after that V_C collapses onto V_D, the result will be an excessively long edge $\overline{V_A V_D}$.

across the line connecting the mid-point of the marked edge and the opposite vertex. This operation is called bisection and an example of it can be seen in Figure 2.9 (left shape).

2. When two edges are marked for refinement, the element is divided into three new elements. This case is shown in Figure 2.9 (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 Figure 2.9 (right shape).

An algorithm for refinement has been proposed by Li *et al.* [71] and can be seen in Algorithm 3. The process starts by initializing a list of edges marked for refinement. The algorithm iterates over all mesh edges and marks for refinement every edge longer than a user-defined maximum length L_{max} (all lengths are calculated in metric space). The point $\mathbf{x_c}$ at which the edge should be split is the middle point in metric space which can be calculated as follows:

$$\mathbf{x_c} = \mathbf{x_0} + \frac{1}{1 + \sqrt{\frac{h_1}{h_0}}}(\mathbf{x_1} - \mathbf{x_0}), \tag{2.12}$$

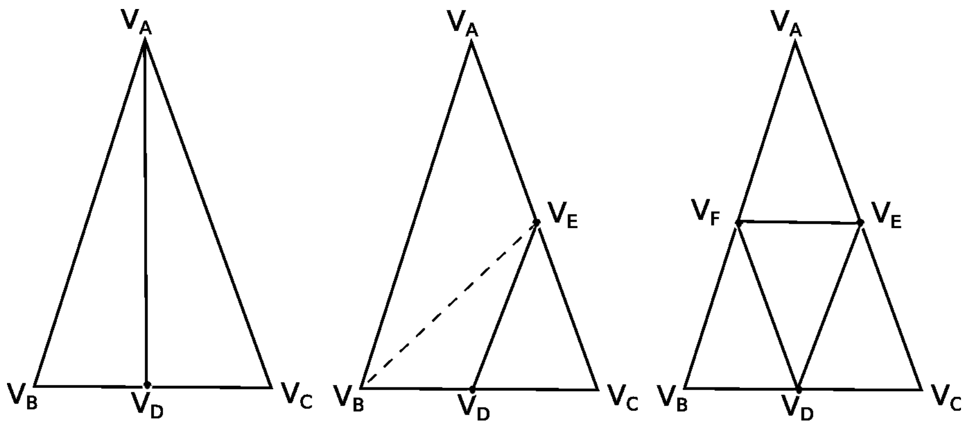


Figure 2.9: Mesh resolution can be increased either by bisecting an element across one of its edges (1:2 split, left figure), by performing a 1:3 split (middle figure) or by performing regular refinement to that element (1:4 split, right figure).

where h_0 and h_1 are the desired edge lengths along $\vec{e}$ at the two ends $\mathbf{x_0}, \mathbf{x_1}$ of the edge, where $\vec{e}$ is the unit vector in the direction of the edge. After $\mathbf{x_c}$ has been calculated, a new vertex $V(\mathbf{x_c})$ is created and appended to the mesh.

After this step, the algorithm loops over all mesh elements. For each element, the algorithm examines how many of the element's edges have been marked for refinement. As was described previously, there can be three discrete cases, each one being dealt with in a different way. Depending on the case, the element is split into two, three or four new elements which are appended to the mesh and the parent element is removed.

In other algorithms, when an element is refined the newly created vertices are non-conforming and this non-conformity must be eliminated by propagating the operation to neighbouring elements. By processing all edges first and then refining each element according to the number of marked edges, this algorithm eliminates the need for propagation.

2.6.3 Edge Swapping

Swapping is a local optimization technique which is used to improve mesh quality by replacing low-quality elements with better ones. The total number of mesh elements remains the same, so there is not subsequent penalty

Algorithm 3 Serial refinement algorithm by Li *et al.* [71].

```
initialize a list of edges for refinement $L_R \leftarrow \emptyset$
loop over all mesh edges
    if length of edge $E_i > L_{max}$ then
        calculate where $E_i$ should be split and create a new vertex $V_i$ at that point
        append $E_i$ to $L_R$
    end if
end loop
loop over all mesh elements
    let $T^i$ be the current element
    let $refine\_cnt$ be the number of element edges $E^i_j \in L_R$
    switch $refine\_cnt$ do
        case 0
            no refinement, continue to next element
        case 1                                          ▷ perform bisection
            let $E^i_0$ be the edge marked for refinement at new vertex $V^i_0$
            create a new edge between $V^i_0$ and the vertex opposite $E^i_0$
            add the two newly created elements $T^i_0$ and $T^i_1$ to the mesh
        case 2                              ▷ perform refinement using diagonals
            let $E^i_0$ and $E^i_1$ be the edges marked for refinement
            let $V^i_0$ and $V^i_1$ be the corresponding new vertices on $E^i_0$ and $E^i_1$
            let $V^i_A$ resp. $V^i_B$ be the vertices opposite $E^i_0$ resp. $E^i_1$
            create new edge $\overline{V^i_0 V^i_1}$
            create diagonal edge $\overline{V^i_0 V^i_A}$ or $\overline{V^i_1 V^i_B}$, whichever is shorter
            add the three newly created elements $T^i_0, T^i_1$ and $T^i_2$ to the mesh
        case 3                                ▷ perform regular refinement
            let $E^i_0$, $E^i_1$ and $E^i_2$ be the edges comprising $T^i$
            let $V^i_0, V^i_1$ and $V^i_2$ be the corresponding new vertices
            create new edges $\overline{V^i_0 V^i_1}$, $\overline{V^i_1 V^i_2}$ and $\overline{V^i_2 V^i_0}$
            add the four new elements $T^i_0, T^i_1, T^i_2$ and $T^i_3$ to the mesh
    remove $T^i$ from the mesh
end loop
```

on the computational cost.

In 2D, swapping is done in the form of edge flipping, i.e. flipping an edge shared by two elements. The operation can be seen in Figure 2.10. The common edge is flipped, resulting in the deletion of two original mesh triangles and their replacement with two new ones. An edge is flipped only if doing so improves the quality of the local mesh patch.

As was the case in coarsening, swapping can be propagated for the purpose of re-evaluating which edges are candidates for flipping after local mesh topology has been altered. An example demonstrating the need for propagation in shown in Figure 2.11. After an edge has been flipped, local topology is modified and adjacent edges which were not considered for flipping before are now candidates. At the end, this procedure results in a *Delaunay Triangulation*, a triangulation in which the minimum element angle in the mesh is the largest possible one with respect to all other triangulations of the mesh [68].

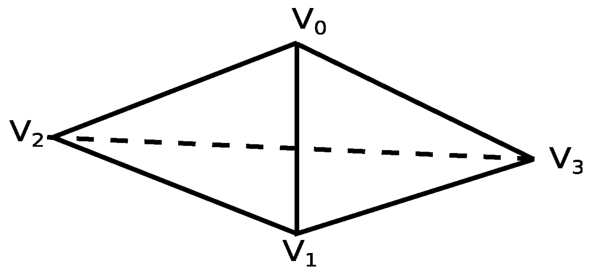


Figure 2.10: Example of edge swapping. Flipping the common edge $\overline{V_0V_1}$ results in the removal of triangles $\widehat{V_0V_1V_2}$ and $\widehat{V_0V_1V_3}$ and their replacement with new triangles $\widehat{V_0V_2V_3}$ and $\widehat{V_1V_2V_3}$.

An algorithm for edge swapping has been proposed by Li *et al.* [71]. The algorithm operates on a list of candidate-edges. The goal is to flip all edges shared by elements which (elements) are of lower quality than a user-defined minimum Q_{min}.

The 2D version of swapping which is implemented in PRAgMaTIc is described in Algorithm 4. The process starts by initializing a list L_S of candidate edges. The algorithm iterates over all mesh edges and adds to L_S every edge which is shared by two elements the quality of which (the quality of the worst element) is lower than Q_{min}. After this step, the algorithm loops over the dynamic list, each time popping a candidate-edge. If the worst element in the local patch is of higher quality than Q_{min}, then there is no need to flip that edge. Otherwise, the algorithm tests whether flipping the edge will improve the worst element quality. If this is the case, then the original edge and elements are removed from the mesh, while the flipped edge and the newly created triangles are appended. Additionally, the four edges outlining the local patch are pushed into the dynamic list, therefore

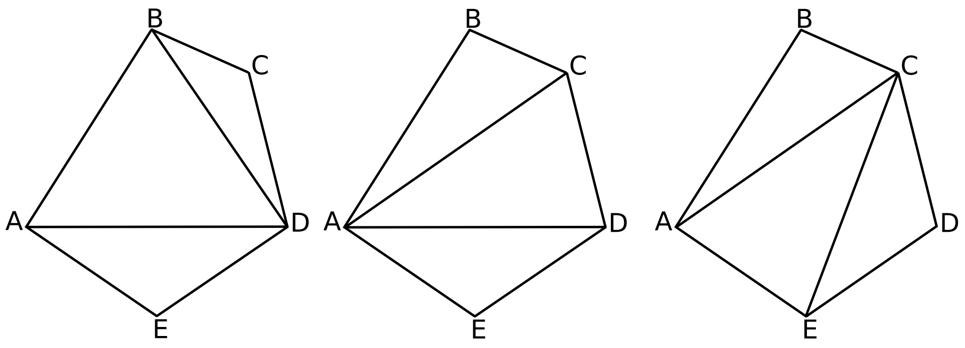


Figure 2.11: Example of swapping propagation. Initially (left figure), edge $\overline{AD}$ is not considered a candidate for swapping because the hypothetical triangles $\widehat{ABE}$ and $\widehat{BDE}$ are of poorer quality than the original triangles $\widehat{ABD}$ and $\widehat{ADE}$. Edge $\overline{BD}$, on the other hand, can be flipped, resulting in improved quality of the local patch (middle figure). After this step, edge $\overline{AD}$ becomes a candidate for swapping, as new elements $\widehat{ACE}$ and $\widehat{CDE}$ are indeed of higher quality than the original elements $\widehat{ABD}$ and $\widehat{ADE}$(right figure).

propagating the operation.

2.6.4 Vertex Smoothing

Smoothing is a crucial component of many unstructured mesh adaptivity algorithms. This provides a powerful, if heuristic, approach to improve mesh quality. A diverse range of approaches to mesh smoothing have been proposed [36, 92, 19, 39, 8, 18, 38, 89]. Effective algorithms for parallelising mesh smoothing extracting concurrency have also been proposed within the context of a Parallel Random Access Machine (PRAM) computational model [37].

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 2.12). Probably the best known heuristic for mesh smoothing is Laplacian smoothing, first proposed by Field [36]. This method operates by moving a vertex to the barycentre of the set of vertices connected by a mesh edge to the

Algorithm 4 Serial edge swapping algorithm by Li *et al.* [71].

```
initialize a list of edges for swapping $L_S \leftarrow \emptyset$
loop over all mesh edges $E^i$
    find elements $T_0^i$ and $T_1^i$ sharing $E^i$
    $Q^i \leftarrow \min(quality[T_0^i], quality[T_1^i])$
    if $Q^i < Q_{min}$ then
        append $E_i$ to $L_S$
    end if
end loop
while $L_S \neq \emptyset$ do
    choose an edge $E^i \in L_S$, $E^i = \overline{V_A^i V_B^i}$
    remove $E^i$ from $L_S$
    find elements $T_0^i = \widehat{V_A^i V_B^i V_0^i}$, $T_1^i = \widehat{V_A^i V_B^i V_1^i}$ sharing $E^i$
    $Q^i \leftarrow \min(quality[T_0^i], quality[T_1^i])$
    if $Q^i > Q_{min}$ then          ▷ no need to consider this patch for swapping
        continue                                ▷ proceed to next edge in $L_S$
    end if
    let $E^j = \overline{V_0^i V_1^i}$                                     ▷ the flipped edge
    let $T_A^j = \widehat{V_A^i V_0^i V_1^i}$, $T_B^j = \widehat{V_B^i V_0^i V_1^i}$                ▷ resulting elements
    $Q^j \leftarrow \min(quality[T_A^j], quality[T_B^j])$
    if $Q^j > Q^i$ then               ▷ if quality of worst element is improved
        remove $E^i$, $T_0^i$ and $T_1^i$ from mesh
        append $E^j$, $T_A^j$ and $T_B^j$ to mesh
        append edges $\overline{V_A^i V_0^i}$, $\overline{V_B^i V_0^i}$, $\overline{V_A^i V_1^i}$, $\overline{V_B^i V_1^i}$ to $L_S$      ▷ propagation
    end if
end while
```

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.* [89]). Therefore, in general, the proposed new position of a vertex $\vec{v}_i^{\mathcal{L}}$ is given by:

$$\vec{v}_i^{\mathcal{L}} = \frac{\sum_{j=1}^{J} ||\vec{v}_i - \vec{v}_j||_M \vec{v}_j}{\sum_{j=1}^{J} ||\vec{v}_i - \vec{v}_j||_M}, \tag{2.13}$$

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 [36], the application of pure Laplacian smoothing can

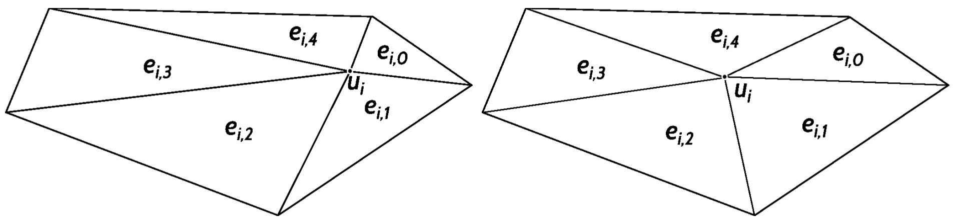


Figure 2.12: Smoothing in a local mesh patch: $\vec{v}_i$ is the vertex being relocated; $\{e_{i,0},\dots,e_{i,m}\}$ is the set of elements connected to $\vec{v}_i$. Smoothing is the operation which relocates $\vec{v}_i$ to a new position so that quality of $\{e_{i,0},\dots,e_{i,m}\}$ is improved.

actually decrease useful local element quality metrics; at times, elements can even become inverted. This can happen if the vertex is relocated outside its *interior convex hull*, *i.e.* the area within which the relocation has to be restricted. An example of an interior convex hull for a mesh vertex can be seen in Figure 2.13. 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.* [39] (while Freitag *et al.* only discuss this for Euclidean geometry it is straightforward to extend to the more general case), is summarised in Algorithm 5. 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):

$$Q(\vec{v}_i) \equiv \|\boldsymbol{q}\|_\infty, \tag{2.14}$$

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

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 6. For this it is assumed that the derivatives of non-inverted element quality are smooth, although the patch quality given in equation (2.14) is not. Note that while $q^M(\triangle)$ as defined in equation (2.8) is not differentiable everywhere, it is differentiable almost everywhere (as F is not differentiable at x=1). The algorithm proceeds by

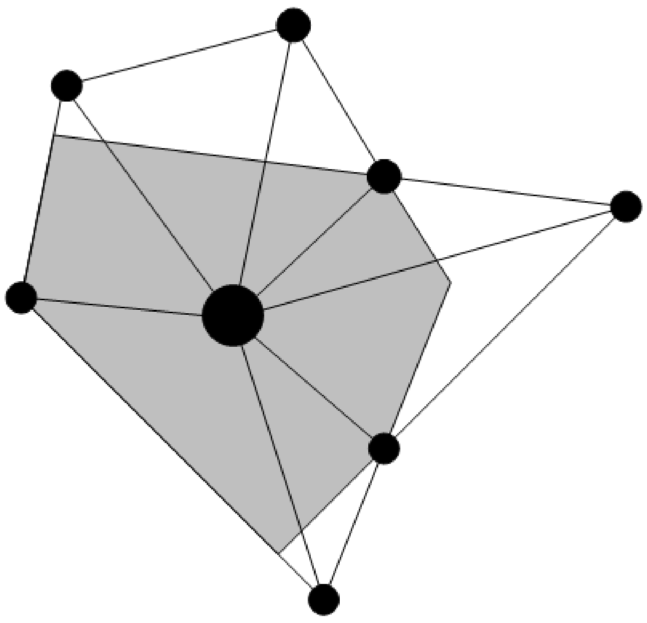


Figure 2.13: Example of an interior convex hull of a vertex. If the vertex under consideration is relocated outside the grey zone, some elements will be inverted and the mesh will be invalid.

stepping in the direction of the quality gradient of the worst element, $\vec{s}$. 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:

$$q' = q + \alpha \vec{\nabla} q \cdot \vec{s}. \tag{2.15}$$

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

$$q' = q + \alpha |\vec{\nabla} q|. \tag{2.16}$$

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

$$q'_e = q_e + \alpha \vec{s} \cdot \vec{\nabla} q_e. \tag{2.17}$$

Algorithm 5 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
$\vec{v}_i^n \leftarrow \vec{v}_i^{\mathcal{L}}$          ▷ Calculate new vertex location using Laplacian smooth and
$M_i^n \leftarrow metric\_interpolation(\vec{v}_i^n)$     ▷ interpolate metric tensor.
$quality^n = Q(\vec{v}_i^n)$     ▷ Calculate the new local quality for this relocation.
          ▷ Loop for max number of iterations or until improvement is made.
**while** $(n \leq max\_iteration)$**and**$(quality_i^n - quality_i^0 < \sigma_q)$ **do**
    $\vec{v}_i^{n+1} \leftarrow (\vec{v}_i^n + \vec{v}_i^0)/2$
    $M_i^{n+1} \leftarrow metric\_interpolation(\vec{v}_i^{n+1})$
    $quality^{n+1} \leftarrow Q(\vec{v}_i^{n+1})$
    $n = n + 1$
**end while**
**if** $quality_i^n - quality_i^0 > \sigma_q$ **then**           ▷ If mesh is improved
    $\vec{v}_i \leftarrow \vec{v}_i^n$                            ▷ update vertex location
    $M_i \leftarrow M_i^n$                          ▷ and metric tensor for that vertex
**end if**
```

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:

$$\alpha = \frac{q - q_e}{\vec{s} \cdot \vec{\nabla} q_e - |\vec{\nabla} q|}. \tag{2.18}$$

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.

2.7 Summary

This chapter provided a brief overview of anisotropic mesh adaptivity. We described some elementary concepts about the Finite Element Method and the need to control solution error, highlighting the context in which adaptivity is used. The special case of anisotropic problems was explained in more detail, with examples of what a metric tensor is and how it is used to

Algorithm 6 Optimisation based smoothing kernel: local element quality is improved by solving a local non-smooth optimisation problem.

```
    ▷ Apply initial smart Laplacian smooth to improve starting position.
$smart\_smooth\_kernel(\vec{v}_i, M_i)$
$quality^0 \leftarrow Q(\vec{v}_i)$
$n \leftarrow 0$
repeat        ▷ Hill climbing until no further improvement in local quality
    $\{\vec{\nabla} q_{e_0}, ..., \vec{\nabla} q_{e_j}, ...\}$        ▷ Calculate initial element quality gradients.
    $\vec{s}^n = \widehat{\nabla \vec{q}_{e_j | q_{e_j} \equiv Q(\vec{v}_i)}}$        ▷ Choose search direction to be that of the
                                        ▷ quality gradient of the worst local element.
    $\alpha = nearest\_discontinuity()$        ▷ Calculate $\alpha$ using equation (2.18).
    $\vec{v}_i^{n+1} = \vec{v}_i^n + \alpha \vec{s}^n$        ▷ Propose new location for vertex.
    $M_i^{n+1} \leftarrow metric\_interpolation(\vec{v}_i^{n+1})$ ▷ Interpolate metric tensor and
    $quality^{n+1} \leftarrow Q(\vec{v}_i^{n+1})$    ▷ evaluate local quality using that location.
    if $quality_i^{n+1} - quality_i^n > \sigma_q$ then        ▷ If the improvement is $> \sigma_q$
        $\vec{v}_i \leftarrow \vec{v}_i^n$        ▷ accept proposed location
        $M_i \leftarrow M_i^n$        ▷ and update metric tensor.
    end if
    $n = n + 1$
until $(n \geq max\_iteration)$or$(quality_i^n - quality_i^{n-1} < \sigma_q)$
```

calculate distances in a generalised space. We analysed the objective functional by Vasilevskii and Lipnikov, which is used as a measure of element quality. At the end we presented the general mesh adaptation procedure and listed algorithms for each optimisation phase, namely coarsening, refinement, swapping and smoothing.

In the following chapter we are about to discuss related work on parallel anisotropic mesh adaptivity and similar *morph algorithms*, *i.e.* algorithms which modify irregular data in non-trivial ways by mutating the relationships between them.

3 Related Work

In this chapter we discuss related work on mesh adaptivity, the primary objective of this research, and auxiliary techniques regarding parallel worklists and optimistic execution used in the scope of general *morph algorithms*. Related work on further techniques (graph colouring and work-stealing schedulers) is presented in the respective Chapters 5 and 7.

3.1 Mesh Adaptivity

There are a number of examples where adaptive mesh methods have been extended to distributed memory parallel computers. The main challenges in performing mesh adaptation in parallel is maintaining a consistent mesh across domain boundaries as well as load-balancing the adapted mesh.

One approach, proposed by Coupez *et al.*, 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 [28]. Parallel efficiency of this approach drops quickly, reaching 18% when using 32 processors on a 2D mesh.

Freitag *et al.* [37, 40] considered 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 the approach described by Alauzet *et al.* [6], 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.

Southern *et al.* propose a strategy similar to Coupez *et al.*. The key difference is that, whereas in Coupez *et al.* previously locked regions are migrated from one processor to another, in Southern *et al.* the whole mesh is repartitioned in such a way that regions requiring further adaptation do not fall on the partition boundary and consequently a processor is free to work on them using the serial adaptive algorithms. This problem is equivalent to minimising the edge-cut after having assigned appropriate weights to edges in regions where further processing is needed. Partitioning and load balancing is achieved using the ParMETIS graph partitioning library [63].

Lipnikov and Vassilevski follow an entirely different approach [72]. The mesh is not distributed among participating processors; instead, the entire mesh is made known to everyone. Decomposition is executed serially on the root processor and adaptation is done on parallel, with each processor working on the sub-domain assigned by the root. Finally, the mesh is re-gathered onto the root processor and this iterative procedure is repeated to convergence. The authors admit that their approach is not scalable and communication dominates computation with as few as 8 processors.

Lepage *et al.* [70] discuss the drawbacks of distributed-memory parallel designs in comparison with shared-memory counterparts and present a modified version of Coupez *et al.* in which vertex smoothing operations are terminated if any of the participating processors has finished its adaptive step. This design compromises mesh quality for the sake of scalability. The authors demonstrate a parallel efficiency of 68% on 8 processors.

3.2 General Morph Algorithms and Amorphous Data Parallelism

Adaptive mesh algorithms fall into the broader group of general *morph algorithms*, which *morph* the underlying data structures in non-trivial ways by adding/removing nodes and edges and mutilating their adjacency lists. Some examples of morph algorithms are investigated in [84]:

- Survey Propagation, a heuristic SAT solver based on Bayesian inference.
- Points-to Analysis, a key static compiler technique.
- Boruvkas Minimum Spanning Tree Algorithm, which computes a minimum spanning tree of an undirected graph using edge contraction.

This publication pertains mainly to GPUs, however the same work is applicable to multicore and manycore platforms as well. Other morph algorithms have been studied throughout the literature, but the ones mentioned above (alongside mesh adaptivity) are the most general in nature and exhibit important challenges.

Morph algorithms are examples of irregular programs exhibiting *amorphous data parallelism* [94, 95], *i.e.* a generalised form of data parallelism which cannot be extracted using static analysis (*e.g.* in compile-time, as a preprocessing step etc.) but can be uncovered in runtime. There have been efforts toward exploiting amorphous parallelism for specific applications, like breadth-first graph traversal on GPUs [82].

In recent years, there is a lot of ongoing research on domain-specific languages and libraries with high-level constructs which allow developers to describe their algorithms intuitively while exposing the inherent data parallelism. Green-Marl [55] is an example of a domain-specific language for graph-data based algorithms. The Green-Marl compiler translates high-level algorithmic descriptions into OpenMP C++ code, exploiting the data-parallelism exposed via the high-level descriptions. SNAP [77] is an open-source graph framework aiming at the study and partitioning of large-scale networks. GraphLab [74, 73] is an abstraction for machine learning and data-mining applications which expresses dynamic, graph-parallel computation while ensuring data consistency and high performance, both in shared-memory environments and distributed systems.

The aforementioned frameworks/DSLs assume that the relationship/connectivity of the underlying irregular data is constant, *i.e.* graph topology is immutable throughout the execution of an algorithm. Morph algorithms have different requirements which dictate the use of more versatile solutions.

Pregel [78] is a framework for processing large graphs in a vertex-centric way, while partially supporting mutation of graph topology. Pregel's restriction is that mutations have to be local, *i.e.* a vertex can add/remove its own outgoing edges or remove itself from the mesh; no other topology modifications are allowed. Therefore, this framework is not as powerful as we need for general morph algorithms.

Galois [51, 95] is a general-purpose system for shared-memory machines which can exploit amorphous data-parallelism in irregular codes that are organised around pointer-based data structures, including graphs. It provides support for general morph algorithms, *i.e.* graph topology can be mutated in every way indicated by the algorithm. Galois uses an irregular compute methodology similar to the one we have developed in this research for parallel worklist algorithms, *i.e.* algorithms in which work items are obtained from a list and new tasks generated from the processing of a work item are added to the list. In Section 4.7 we highlight the basic differences between Galois and our framework.

3.3 Summary

In this chapter we presented related work in the field of parallel anisotropic mesh adaptivity and concluded that most efforts target distributed-memory parallelism, leaving a gap to be filled for mesh adaptivity in the manycore era. Following that, we reviewed efforts around frameworks, domain-specific languages and libraries for parallelising algorithms with irregular data and exploiting amorphous parallelism. In the following chapter we are going to present an irregular compute methodology which was developed to assist us in developing parallel adaptive mesh algorithms for shared-memory systems.

4 Irregular Compute Methodology

In this chapter we discuss our design choices and the techniques we have used which allow safe and scalable parallel execution of algorithms on unstructured data. Although anisotropic mesh adaptivity is the test case we used, the same compute methodology can be generalised for other applications with unstructured data.

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.* [40]. 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. Propagation of adaptive operations is assisted by the use of parallel worklists, manipulated with atomics.

4.1 Parallel Execution Framework

Trying to ensure data consistency is one of the main reasons why parallel execution performance can be hindered. Defining tasks that can execute concurrently is challenging because of the complex data dependencies. Locks and synchronisations need to be avoided where possible because they can severely degrade the scalability of mesh adaptivity.

Freitag *et al.* [40] introduced the concept of elemental operations (or compute kernels, using modern terminology) and proposed that data consistency is maintained only in between successive executions of these operations (and not during the execution). This requirement leads to the formulation of the elemental operation steps:

(a) Parallel execution of a set of some mesh improving techniques in each participating processing unit.

(b) Global reduction between these units to update data modified by (a).

Retaining data consistency comprises three requirements. The first one pertains to the uniqueness of data ownership; there cannot exist two processing units sharing ownership of the same data. In the context of shared-memory (intra-node) parallelism, there is no notion of ownership since all logical threads have read/write access to the entire mesh. Secondly, every mesh vertex must have complete knowledge of its neighbouring vertices, *i.e.* their IDs and their coordinates on the mesh. Neighbouring relationship has to be reciprocal, *i.e.* if processor P_1 owns vertex V_1 and $P1$ knows that V_1 is adjacent to V_2 owned by processor P_2, then P_2 also has to know that V_2 is V_1's neighbour. Finally, there is an analogous requirement for reciprocal knowledge of element adjacency.

The execution of an adaptive algorithm can be modelled with the use of an operation task graph $\mathcal{G}$. The elemental operations comprising the execution of the algorithm are represented as vertices of $\mathcal{G}$. An edge connects two vertices if the respective elemental operations depend on each other. Using the task graph allows us to extract independent sets of operations that can be safely executed in parallel. Independent sets can be obtained by using some graph colouring algorithm. After processing an independent set, a global reduction takes place so that updated adjacency information is circulated among neighbouring processors. At this point it is guaranteed that the distributed data structure is consistent, since all operations executed were independent from each other.

The general algorithm is summarised in Algorithm 7. This algorithm consists of two loops. The outer loop is call the propagation loop, because it spawns new elemental operations. As was described earlier, propagation is necessary because topological changes to a local mesh patch might give rise to new configurations of better quality (see Figure 2.11). As for the inner loop, the number of iterations performed depends on the task graph and, more importantly, the way independent sets are extracted from it. The nature of adaptive algorithms used in this project implies that the elemental operations can run asynchronously and mostly require only one-to-one communication between processing units (for other optimisation algorithms a few global reductions would also be required). This property is very important in the scope of efficiency and scalability of a parallel application.

Algorithm 7 General parallel algorithm by Freitag *et al.* [40] for mesh adaptation.

```
$\mathcal{G} \leftarrow$ the operation task graph
while $\mathcal{G} \neq \emptyset$ do
    colour $\mathcal{G}$
    initialize new set $\mathcal{R} \leftarrow \emptyset$
    for all independent sets $\mathcal{I}_j$ of $\mathcal{G}$ do
        execute all operations of $\mathcal{I}_j$ in parallel
        $\mathcal{R} \leftarrow \mathcal{R} \cup$ {new elemental operations spawned by processing $\mathcal{I}_j$}
    end for
    $\mathcal{G} \leftarrow \mathcal{R}$
end while
```

4.2 Design Choices

4.2.1 Threading Mechanism

In view of the switch to multi-core nodes, adaptive mesh methods based on traditional task-based parallelism (as described in Section 4.1) 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 fully exploit the shared memory within a node. OpenMP is our preferred choice due to its greater potential for use with co-processors such as Intel®Xeon Phi™ [66] and its simpler interface (via `#pragma` directives in C code), that simplifies code maintenance, while there is excellent support by various toolsets (*e.g.* profilers and debuggers).

4.2.2 Processor affinity

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 CPUs. 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 data from memory nodes which are not directly connected to the CPU and cache misses.

4.2.3 Mesh Data Structures

The minimal information required to represent a mesh is an element-node list (referred to as `ENList`), which is implemented in PRAgMaTIc as an STL vector of triplets of vertex IDs (`std::vector<int>`), and an array of vertex coordinates (referred to as `coords`), which is an STL vector of pairs 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.

It is also necessary to store the metric tensor field. The field is discretised node-wise and every metric tensor is a symmetric 2-by-2 matrix. For each mesh node, we need to store three values for the tensor: two values for the two on-diagonal elements and one value for the two off-diagonal elements. Thus, `metric` is an STL vector of triplets of metric tensor values (`std::vector<double>`). The three components of the metric at vertex `vid` are stored at `metric[3*vid]`, `metric[3*vid+1]` and `metric[3*vid+2]`.

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`). `NNList` is implemented as an STL vector of STL vectors of vertex IDs (`std::vector< std::vector<int> >`). `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 source of overhead. Instead, we opted for the approach of finding all necessary adjacency information on the fly using `ENList`, `NNList` and `NEList`.

4.3 Topological Hazards and Vertex Colouring

There are two types of hazards when running adaptive algorithms in parallel: topological (or structural) hazards and data races. The term topological hazards refers to the situation where an adaptive operation results in invalid or non-conforming edges and elements. An example can be seen in Figure 4.1. If two threads flip edges $\overline{AD}$ and $\overline{BD}$ at the same time, the result will be two new edges $\overline{AC}$ and $\overline{BE}$ crossing each other. Structural hazards can be avoided by colouring mesh vertices and processing them in batches of independent sets. A discussion on data races and how they can be avoided follows in Section 4.4.

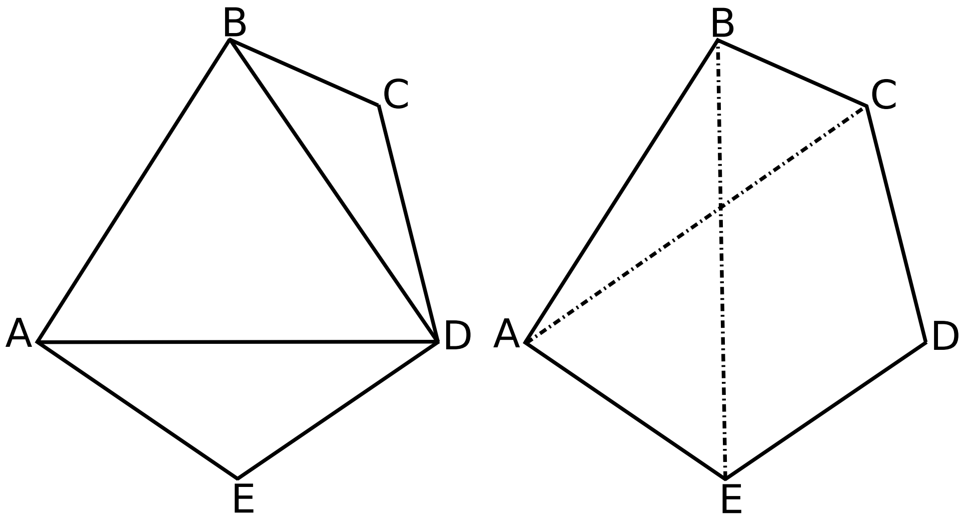


Figure 4.1: Example of topological hazard when running adaptive mesh algorithms in parallel. If two threads flip edges $\overline{AD}$ and $\overline{BD}$ at the same time, the result will be an invalid local mesh patch in which two edges $\overline{AC}$ and $\overline{BE}$ cross each other.

Inspired by Freitag's parallel framework, we designed our implementations so that all elemental operations are operations on vertices, i.e. the task graph $\mathcal{G}$ is the mesh itself and parallel adaptivity is achieved via assigning all vertices of an independent set to participating threads. This saves us from constructing and maintaining other complex data structures, like edge-edge and element-element adjacency lists. In coarsening and smoothing this choice is straightforward to understand, since those algorithms operate directly on vertices (for removal or relocation). For refinement and swapping, which operate on edges, we follow the convention that an edge may only be processed (split or flipped) by the thread which has been assigned the vertex with the lesser ID.

The fact that topological changes are made to the mesh means that the graph colouring is frequently invalidated and the mesh has to be re-coloured before proceeding to the next iteration of the adaptive algorithm. Therefore, a fast scalable graph colouring algorithm is vital to the overall performance. In this work we have developed a parallel colouring algorithm based on previous work by Çatalyürek *et al.* [20]. In Chapter 5 we discuss and compare a number of parallel graph colouring techniques and present how the new improved algorithm was devised.

4.4 Data Races and Deferred Updates

Data races in adaptive mesh algorithms can appear when two or more threads try to update the same adjacency list. An example can be seen in Figure 4.2. Having coloured the mesh, two threads are allowed to process vertices V_B and V_D at the same time and without structural hazards. One thread T_0 coarsens edge $\overline{V_B V_C}$ and vertex V_B collapses on V_C. `NNList[` V_C `]` and `NEList[` V_C `]` are modified by T_0, *e.g.* V_A must be added to `NNList[` V_C `]`. At the same time, another thread T_1 coarsens edge $\overline{V_D V_C}$ and vertex V_D collapses on V_C. `NNList[` V_C `]` and `NEList[` V_C `]` are modified by T_1 as well, *e.g.* V_E must be added to `NNList[` V_C `]`. Both threads try to update `NNList[` V_C `]` and `NEList[` V_C `]` at the same time, so there is a data race which could lead to data corruption.

One solution could be a 2-distance colouring of the mesh (a d-distance colouring of $\mathcal{G}$ is a colouring in which no two vertices share the same colour if these vertices are up to d edges away from each other or, in other words, if there is a path of length $\leq d$ from one vertex to the other [35]). Although this solution guarantees a race-free execution, calculating a 2-distance colouring is expected to take more time than a simple 1-distance colouring while using

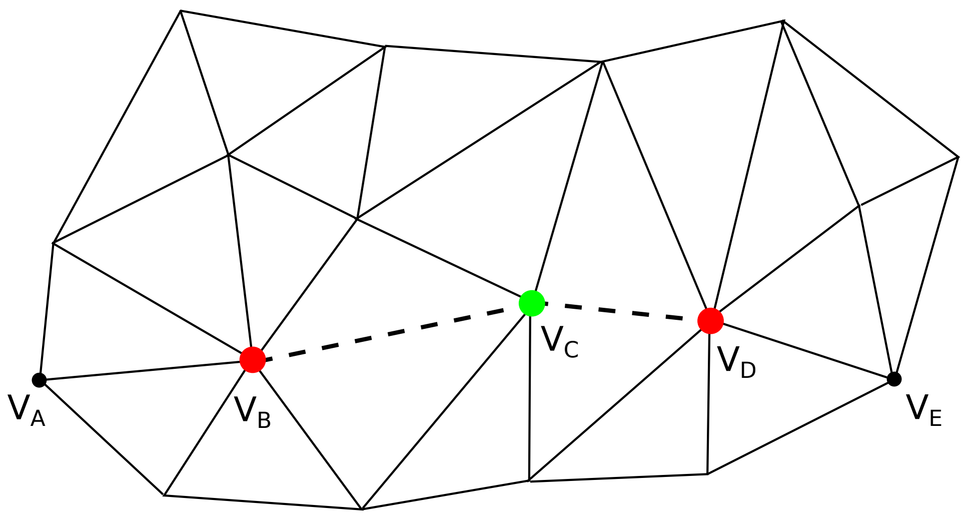


Figure 4.2: Example of data races when trying to update adjacency lists in parallel. Only colouring the mesh is not enough to guarantee race-free execution.

a higher number of colours. We opted for another approach, which we call "deferred updates mechanism".

In an shared-memory environment with `nthreads` OpenMP threads, every thread has a private collection of `nthreads` lists, one list for each OpenMP thread (see Figure 4.3). 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 $tid = i\%nthreads$. After processing an independent set (recall that every algorithm is organised as a series of adaptive sweeps through independent sets) and before proceeding to the next one, 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 Code Snippet 1. It can be said that this mechanism is our way of implementing Freitag's proposal that "data consistency is maintained only in between successive executions of an adaptive

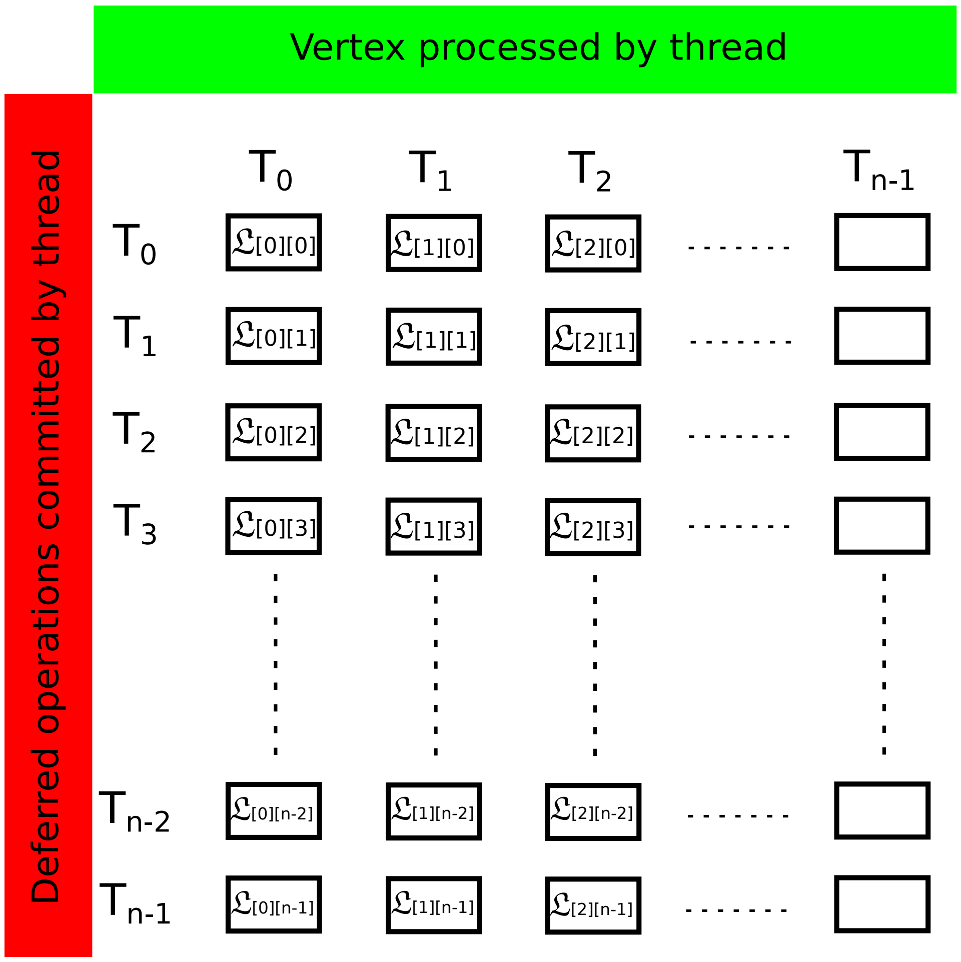


Figure 4.3: Schematic depiction of the deferred updates mechanism. Every thread has a private collection of *nthreads* lists. Updates from thread T_U pertaining to vertex i are pushed back into list $\mathcal{L}[T_U][T_C]$, where $T_C = i\%nthreads$ is the thread responsible for committing the updates for that vertex. After processing an independent set, every thread T_C visits all lists $\mathcal{L}[0..n-1][T_C]$ and commits the operations stored on them.

algorithm and not during the execution".

An important advantage of the deferred updates strategy is that it does not lead to differences in quality of the final mesh compared to an "as we go" write-back scheme. By committing the updates at the end of every independent set, we always use the most up-to-date information. Not

```
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.

  // Process one independent set in parallel
  // Defer any updates until the end of the for-loop
  #pragma omp for
  for(int i=0; i<nVerticesInSet; ++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]);
  }

  // Proceed to the next independent set...
}
```

Code Snippet 1: Typical example of using the deferred updates mechanism

using stale data eliminates the risk of mesh data corruption in coarsening, refinement and swapping, whereas in smoothing we have a faster-converging Gauss-Seidel-style iteration process. Additionally, memory footprint of this mechanism is negligible. Complexity in terms of memory is $\Theta\left(\textit{vertices in independent set}\right)$.

4.5 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 Sections 6.1 and 6.3, respectively. 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 [11] (also known as inclusive scan or partial reduction in MPI terminology) 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 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® and Intel®Xeon Phi™, 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 Code Snippet 2. A similar atomic-based approach (using atomic compare-and-swap) is used in Galois for certain types of worklists.

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

```
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));

}
```

Code Snippet 2: Example of creating a worklist using OpenMP's atomic capture operations.

atomic-capture operations is insignificant.

4.6 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 used by other research groups); 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.

4.7 Comparison with Galois and Optimistic Execution

Apart from mesh adaptivity, the irregular compute methodology presented in this chapter can be used for general morph algorithms. Our framework is just as powerful as Galois, albeit accomplishing the goals of high performance and safe parallel execution by following a different approach.

The key difference between the two approaches lies in explicitly thread-safe vs optimistic execution. Galois already abstracts worklist manipulation (in fact, it supports more iteration-scheduling policies in addition to linear queues) and provides special data structures which are essential for thread-safe execution under the optimistic (or speculative) model. The idea of speculative execution is that a thread executes the computational kernel as if it were the only worker in the system, without caring about races; if another thread tries to modify data already marked as being locked by the first thread, then a conflict is reported to the runtime system and one of the conflicting activities is reverted (rolled-back). Once the user has written the algorithm using Galois-provided data structures and annotated which loops are to be executed in parallel, "the Galois system then speculatively extracts as much parallelism as it can." [51].

On the other hand, our methodology in its current form explicitly enforces thread safety by using the combination of colouring and the deferred operations mechanism to accomplish race-free parallel execution, without any provision of special data structures to support speculation. Both approaches involve some overhead:

- Optimistic execution: Overhead of rolling-back and re-attempting to apply the computational kernel to a subset of the graph.

- Our framework: Overhead of graph colouring and committing the deferred updates (which involves additional thread synchronisation).

Neither approach seems to be universally superior to the other; we believe that performance of each methodology is highly dependent upon the specific algorithm under consideration as well as properties (e.g. connectivity) of the graph.

4.8 Conclusion

In this chapter we described our methodology for working with irregular data. We listed our design choices regarding mesh data structures, which made maintenance of their consistency easier. We demonstrated what kind of topological hazards there are in adaptive kernels and how the colouring-based parallel framework by Freitag *et al.* ensures that the mesh structure is not invalidated during adaptation. We also gave an overview of potential data races while committing changes to the mesh and showed that the deferred-updates strategy eliminates them. Finally, we discussed our alternative approach to the creation of parallel worklists using atomic fetch-and-add, a technique which has the advantage of being synchronisation-free compared to classic reduction-based approaches.

The importance of graph colouring in adaptive algorithms made us go after a fast and scalable way of colouring the mesh. In the next chapter we will review previous work on parallel graph colouring and present an improved technique which is shown to outperform its predecessors.

5 Graph Colouring and Speculative Execution

Unstructured mesh applications, like anisotropic mesh adaptivity, are being optimised for modern multi-core and many-core architectures. Graph colouring is often an important preprocessing step as a means of guaranteeing safe parallel execution in a shared-memory environment. Examples of such applications include (but are not limited to) iterative methods for sparse linear systems [62], sparse tiling [106], eigenvalue computation [79] and preconditioners [102, 56].

The total run time of a colouring algorithm adds to the overall parallel overhead of the application whereas the number of colours used determines the amount of available parallelism. A fast and scalable colouring algorithm using as few colours as possible is vital for the overall parallel performance and scalability of many unstructured mesh applications. In this chapter we study various parallel colouring techniques and show how we devised an improved version based on speculative execution which runs faster than existing methods while keeping the number of used colours at the same low levels.

5.1 Background and Related Work

The simplest graph colouring algorithm (and one of the most commonly used) is the greedy one, formally known as *First-Fit* colouring (§5.2). There have been efforts towards parallelising the algorithm for shared-memory environments (distributed-memory versions also exist, but studying them is out of scope of this research). Non-greedy techniques like *Largest-Degree-First* [113], *Smallest-Degree-Last* [81], *Saturation-Degree-Ordering* [15] and *Incidence-Degree-Ordering* [24] were not considered here because they either are not well-suited to parallelisation or have worse than linear complexity $(O(n^2), O(n^3))$ and do not minimise the amount of colours sufficiently enough to justify the extra runtime compared to the greedy algorithm [7].

The most notable parallel greedy colouring algorithm is the one by Jones and Plassmann [61] (§5.3), which in turn is an improved version of the original *Maximal Independent Set* algorithm by Luby [75]. A modified version of Jones-Plassmann, presented at the 2012 Nvidia GPU Technology Conference by Jonathan Cohen and Patrice Castonguay [23], uses multiple hashes to minimise thread synchronisation (§5.4).

A different series of parallel greedy colouring algorithms based on speculative execution was introduced by Gebremedhin and Manne [47] (§5.5). Çatalyürek *et al.* presented an improved version of the original algorithm in [20] (§5.6). We took the latter one step further, devising a method which runs under an even more speculative scheme with less thread synchronization (§5.7). Although performance on 2D simplicial meshes is not significantly improved, we show that the new technique can perform considerably faster than its predecessor on 3D simplicial meshes and on highly irregular graphs with high-degree vertices by scaling further on multi-core and many-core systems, while using equally few colours.

5.2 First-Fit Colouring

Colouring a graph with the minimal number of colours has been shown to be an NP-hard problem [46]. For any planar graph (like 2D simplicial meshes), it is known that the chromatic number, *i.e.* the optimal number of colours required to colour it, is 4 [10]. There exist heuristic algorithms which colour a graph in polynomial time using relatively few colours, albeit not achieving an optimal colouring. One of the most common polynomial colouring algorithms is *First-Fit*, also known as *greedy colouring*. In its sequential form, First-Fit visits every vertex and assigns the smallest colour available, *i.e.* not already assigned to one of the vertex's neighbours. The procedure is summarised in Algorithm 8.

It is easy to give an upper bound on the number of colours used by the greedy algorithm. Let us assume that the highest-degree vertex V_h in a graph has degree d, *i.e.* this vertex has d neighbours. In the worst case, each neighbour has been assigned a unique colour; then one of the colours $\{1, 2, \ldots, d + 1\}$ will be available to V_h (*i.e.* not already assigned to a

Algorithm 8 Sequential greedy colouring algorithm.

```
Input: $\mathcal{G}$
for all vertices $V_i \in \mathcal{G}$ do
    $\mathcal{C} \leftarrow \{$colours of all coloured vertices $V_j \in adj(V_i)\}$
    $c(V_i) \leftarrow \{$smallest colour $\notin \mathcal{C}\}$
end for
```

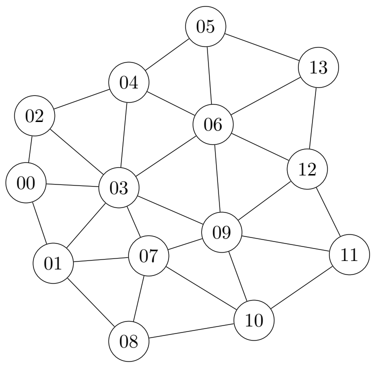
neighbour). Therefore, the greedy algorithm can colour a graph with at most $d + 1$ colours. In fact, experiments have shown that First-Fit can produce near-optimal colourings for many classes of graphs [24].

5.3 Algorithm by Jones-Plassmann

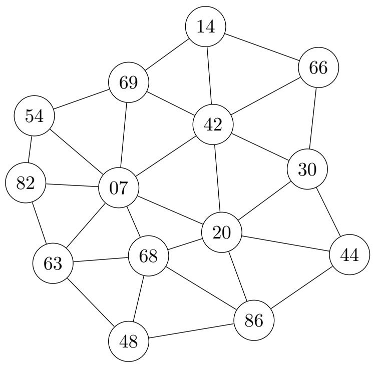
Jones and Plassmann presented a parallel version of First-Fit in [61]. This algorithm is based on Luby's proposal of finding maximal independent sets of vertices and colouring them in parallel [75]. For the Jones-Plassmann algorithm we only need to find independent sets, not necessarily maximal ones. An independent set is constructed in parallel using a Monte Carlo method. Every vertex is assigned a weight. The weights chosen by Luby are the result of some random permutation of vertex IDs. An independent set of uncoloured vertices is then formed in parallel by choosing all vertices whose weights are local maxima, *i.e.* larger than the weight of any uncoloured neighbour.

Vertex IDs are usually dependent on the location of each vertex in the mesh. In order to get a good permutation we need to shuffle the IDs using a hash function $hf(ID)$ which maps location-dependent IDs to random numbers. A hashing function known as a Park and Miller pseudo-random number generator [91], which in turn is based on work by D. H. Lehmer [69], is shown in Code Snippet 3. In this function, n is a large prime number and g is a number of high multiplicative order modulo n [112]. This property of g guarantees that any ID in the range $[0..n)$ will be mapped to a unique number, i.e. no two IDs in that range can have the same hash.

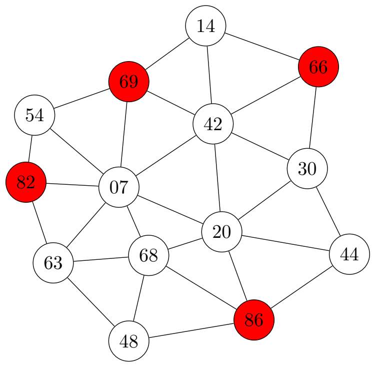
Once an independent set has been found, all vertices in it can be coloured in parallel using the First-Fit principle, *i.e.* they are given the smallest colour not already assigned to a neighbour. This procedure is repeated until all vertices have been coloured. Figure 5.1 shows an example of how



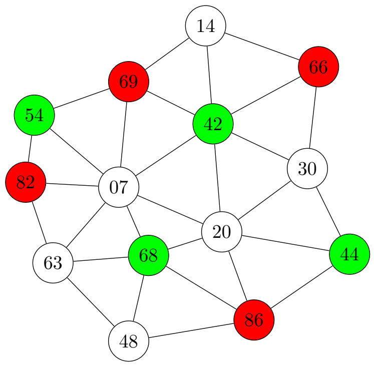
(a) Initial graph



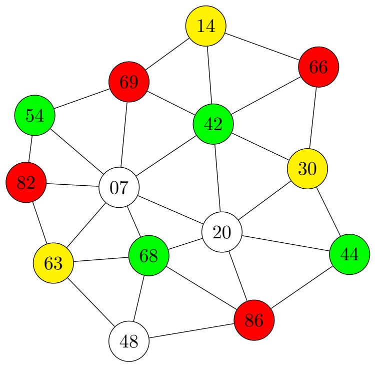
(b) Weights assigned to vertices



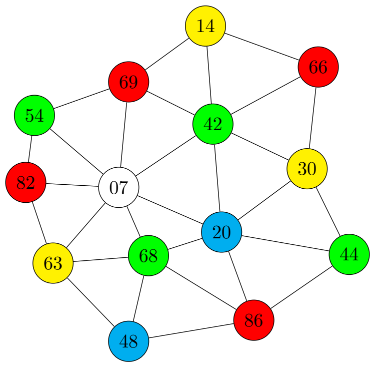
(c) Round 0



(d) Round 1



(e) Round 2



(f) Round 3

Figure 5.1: Example of colouring a graph using the Jones-Plassmann algorithm. In each round, vertices whose weights are local maxima among all uncoloured neighbours are coloured in parallel and are given the smallest colour available.

```c
int hash(int vid){
  const int n = large_prime_number;
  const int g = number_of_high_multiplicative_order_modulo_n;
  return (vid * g) % n;
}
```

Code Snippet 3: Simple hash function using the Park & Miller pseudo-random number generator.

this algorithm progresses. Algorithm 9 summarises the Jones-Plassmann colouring method. As can be seen, there are two thread synchronisation points per iteration of the while-loop. In line with Amdahl's law, it is only expected that scalability will be limited unless thread synchronisation is minimised. In fact, Allwright *et al.* benchmarked this algorithm both on SIMD and MIMD parallel systems and reported no speedup at all [7]. Jones and Plassmann themselves did not claim getting any speedup either [61]; they only mention that "the running time of the heuristic is only a slowly increasing function of the number of processors used".

Algorithm 9 Jones-Plassmann parallel colouring algorithm.

```
Input: $\mathcal{G}(V, E)$
$\mathcal{U} \leftarrow V$                                        ▷ set of uncoloured vertices
while $|\mathcal{U}| > 0$ do
    #pragma omp parallel for
    for all vertices $V_i \in \mathcal{U}$ do
        $\mathcal{I} \leftarrow \{$all $V_i$ for which $w(V_i) > w(V_j)\ \forall V_j \in adj(V_i)\}$
    end for
    #pragma omp barrier                                ▷ synchronise threads

    #pragma omp parallel for
    for all vertices $V_i \in \mathcal{I}$ do                ▷ colour them using First-Fit
        $\mathcal{C} \leftarrow \{$colours of all coloured vertices $V_j \in adj(V_i)\}$
        $c(V_i) \leftarrow \{$smallest colour $\notin \mathcal{C}\}$
        $\mathcal{U} \leftarrow \mathcal{U} - V_i$                    ▷ remove from set of uncoloured vertices
    end for
    #pragma omp barrier                                ▷ synchronise threads
end while
```

5.4 Algorithm by Jones-Plassmann with Multiple Hashes

Taking Jones-Plassmann colouring a step further, we can eliminate the need for thread synchronisation by using more than one weights for vertices. This idea was presented at the 2012 Nvidia GPU Technology Conference by Jonathan Cohen and Patrice Castonguay. One way to compute multiple weights is by using many different hash functions $hf_0(ID)$, $hf_1(ID)$, $\ldots$, $hf_{n-1}(ID)$ or, alternatively, we can keep re-hashing the computed weights multiple times using a single hash function $hf(ID)$, *i.e.* compute $hf(hf(hf(\ldots hf(ID)\ldots)))$. The idea is that we keep computing weights with different hash functions (resp. keep re-hashing the weights) until we reach the point where the vertex under consideration has got the highest weight in its neighbourhood. If the local maximum is reached using $hf_k(ID), k \in [0, n-1)$ (resp. by re-hashing the weight for k times), then the vertex can be immediately assigned colour k, independently from the colours of neighbours.

The whole procedure can be seen in Algorithm 10. A thread colouring a vertex does not examine whether a neighbour has been coloured; it only examines the hashes. Therefore, it is obvious that this version of the Jones-Plassmann algorithm is even more parallel with no thread synchronisation at all. An interesting application of this colouring algorithm would be in a distributed-memory system where some global vertex numbering has been set up beforehand. By hashing global vertex IDs instead of local ones we can get consistent colouring across participating processors with no communication at all.

Algorithm 10 Jones-Plassmann colouring with multiple hash functions.

```
Input: $\mathcal{G}(V, E)$, set of hash functions $hf_0$, $hf_1$, $\ldots$, $hf_{n-1}$
#pragma omp parallel for
for all vertices $V_i \in \mathcal{G}$ do
    $k \leftarrow 0$
      $\triangleright$ keep probing different $hf_k$ while weight($V_i$) is not local maximum
    while $\exists V_j \in adj(V_i)$ for which $hf_k(V_j) > hf_k(V_i)$ do
        $k \leftarrow k + 1$
    end while
    $c(V_i) \leftarrow k$                                  $\triangleright$ assign colour $k$ to vertex
end for
```

On the other hand, the algorithm deviates from the First-Fit principle, since the colour assigned to a vertex is not the smallest available but depends solely on some local configuration of hashes. An immediate consequence is that the upper bound on the number of colours given in §5.2 does not apply in this case. In fact, our experimental results revealed that this algorithm leads to unjustifiably large numbers of used colours, around an order of magnitude higher than the ones delivered by the parallel variants of First-Fit. Having too many colours reduces the amount of exposed parallelism in adaptive mesh algorithms and exaggerates the overhead of thread synchronisation (as was described in Section 4.1, there needs to be some global reduction after processing an independent set, which translates to thread synchronisation in shared-memory systems).

5.5 Algorithm by Gebremedhin-Manne

Gebremedhin and Manne took a different approach to parallelising the greedy graph colouring algorithm. In [47] they describe a fast and scalable greedy colouring technique for shared-memory systems based on the principles of speculative (or optimistic) execution. The idea is that we can colour all vertices in parallel using First-Fit without caring for race conditions at first; this can lead to defective colouring, *i.e.* two adjacent vertices might get the same colour. Defects can be spotted and fixed at a later stage.

The exact method can be seen in Algorithm 11. The process comprises three stages: *pseudo-colouring*, *conflict detection* and *conflict resolution.* During the first stage, every thread colours vertices as if it was working alone, *i.e.* the thread visits a subset of the graph and applies the sequential greedy colouring algorithm to all vertices of that subset without caring about race conditions. Consequently, it is possible that two threads may attempt to colour adjacent vertices simultaneously and give them the same colour, therefore producing an invalid colouring, known as *pseudo-colouring.* In the second stage, threads visit all graph vertices again (in parallel) and check for conflicts. If two adjacent vertices have been assigned the same colour, then one of them (by convention the vertex with the lesser ID) is push back into a list of conflicting vertices. Finally, a single thread resolves conflicts by re-colouring those vertices sequentially in the third stage.

The resulting algorithm is highly parallel, since most of the colouring is

Algorithm 11 Gebremedhin-Manne parallel graph colouring algorithm.

```
Input: $\mathcal{G}(V, E)$
                                        ▷ Stage 1 - Pseudo-colouring (in parallel)
#pragma omp parallel for
for all vertices $V_i \in \mathcal{G}$ do                 ▷ colour them using First-Fit
    $\mathcal{C} \leftarrow$ {colours of all coloured vertices $V_j \in adj(V_i)$}
    $c(V_i) \leftarrow$ {smallest colour $\notin \mathcal{C}$}
end for
#pragma omp barrier
                                        ▷ Stage 2 - Conflict detection (in parallel)
$\mathcal{L} \leftarrow \emptyset$                       ▷ global list of defectively coloured vertices
#pragma omp parallel for
for all vertices $V_i \in \mathcal{G}$ do
    if $\exists V_j \in adj(V_i) : c(V_i) == c(V_j)$ and $id(V_i) < id(V_j)$ then
        $\mathcal{L} \leftarrow \mathcal{L} \cup V_i$                 ▷ mark $V_i$ as defectively coloured
    end if
end for
#pragma omp barrier
                                        ▷ Stage 3 - Conflict resolution (serially)
              ▷ apply the serial greedy algorithm on all conflicting vertices
for all vertices $V_i \in \mathcal{L}$ do
    $\mathcal{C} \leftarrow$ {colours of all coloured vertices $V_j \in adj(V_i)$}
    $c(V_i) \leftarrow$ {smallest colour $\notin \mathcal{C}$}
end for
```

done optimistically in a synchronisation-free and lock-free fashion. There is a major weakness, however, in this algorithm which lies in the explicitly sequential stage at the end. The workload during this stage depends on the amount of defectively coloured vertices. As the number of threads increases, it is only expected that the amount of conflicts increases as well, therefore exaggerating the penalty of the sequential stage on the algorithm's scalability. In their original experiments, Gebremedhin and Manne confirmed this assumption and their benchmarks on various graphs showed a parallel efficiency which drops below 75% when using 12 OpenMP threads. As we enter the manycore era, a more parallel approach has to be pursued.

5.6 Algorithm by Çatalyürek *et al.*

Picking up where Gebremedhin and Manne left off, Çatalyürek *et al.* improved the original algorithm by removing the third sequential stage and applying the first two stages iteratively. This work was presented in [20]. Each of the two phases, called *tentative colouring* phase and *conflict detection* phase respectively, is executed in parallel over a relevant set of vertices. Like the original algorithm by Gebremedhin and Manne, the tentative colouring phase produces a pseudo-colouring of the graph, whereas in the conflict detection phase threads identify defectively coloured vertices and append them into a list $\mathcal{L}$. Instead of resolving conflicts in $\mathcal{L}$ serially, $\mathcal{L}$ now forms the new set of vertices over which the next execution of the tentative colouring phase will iterate. This process is repeated until no conflicts are encountered.

Algorithm 12 The parallel graph colouring algorithm technique by Çatalyürek *et al.*.

```
Input: $\mathcal{G}(V, E)$
$\mathcal{U} \leftarrow V$
while $\mathcal{U} \neq \emptyset$ do
                                        ▷ Phase 1 - Tentative colouring (in parallel)
    #pragma omp parallel for
    for all vertices $V_i \in \mathcal{U}$ do            ▷ colour them using First-Fit
        $\mathcal{C} \leftarrow$ {colours of all coloured vertices $V_j \in adj(V_i)$}
        $c(V_i) \leftarrow$ {smallest colour $\notin \mathcal{C}$}
    end for
    #pragma omp barrier
                                        ▷ Phase 2 - Conflict detection (in parallel)
    $\mathcal{L} \leftarrow \emptyset$                    ▷ global list of defectively coloured vertices
    #pragma omp parallel for
    for all vertices $V_i \in \mathcal{U}$ do
        if $\exists V_j \in adj(V_i) : c(V_i) == c(V_j)$ and $id(V_i) < id(V_j)$ then
            $\mathcal{L} \leftarrow \mathcal{L} \cup V_i$              ▷ mark $V_i$ as defectively coloured
        end if
    end for
    #pragma omp barrier
    $\mathcal{U} \leftarrow \mathcal{L}$            ▷ Set of vertices to be coloured in the next round
end while
```

Algorithm 12 summarises this colouring method. As can be seen, there is no sequential part in the whole process. This trait constitutes a significant improvement over Gebremedhin-Manne in terms of the algorithm's ability to keep scaling as the number of threads rises. Additionally, speed does not come at the expense of colouring quality. The authors have demonstrated that this algorithm produces colourings using about the same amount of colours as the serial greedy algorithm (there is a negligible deviation from the serial results for some graphs). As is stated in their original publication, “the increase in number of colours with an increase in concurrency is none to modest”. However, there is still a source of sequentiality, namely the two thread synchronisation points in every iteration of the while-loop. Thread synchronisation can easily become a barrier for scalability and should be minimised or eliminated if possible.

5.7 An Improved Algorithm

Moving toward the direction of removing as much thread synchronisation as possible, we managed to improve the algorithm by Çatalyürek *et al.* by eliminating one of the two barriers inside the while-loop. This was achieved by merging the two parallel for-loops inside the while-loop into a single parallel for-loop. We observed that when a vertex is found to be defective, it can be re-coloured immediately instead of deferring its re-colouration for the next round. Therefore, the tentative-colouring phase and the conflict-detection phase can be combined into a single *detect-and-recolour* phase. Doing so leaves only one thread synchronisation point in every iteration of the while-loop, as can be seen in Algorithm 13.

It would be ideal to be able to remove all thread synchronisation completely. We believe, however, that this is not possible beyond the point we have reached and we claim that our improved technique presented here is both the fastest and the most scalable among its competitors in the family of parallel greedy graph colouring algorithms, while being one of the best examples of what speculative execution can offer when used in algorithms operating on irregular data structures.

5.8 Experimental Results

In order to evaluate our new colouring method and compare it to the previous state-of-the-art technique by Çatalyürek *et al.* we ran a series of benchmarks using 2D finite element meshes and 3D finite volume meshes, alongside randomly generated graphs using the *R-MAT* graph generation algorithm [21]. Simplicial 2D/3D meshes are used in order to measure per-

Algorithm 13 Our improved parallel graph colouring technique based on the algorithm by Çatalyürek *et al.*.

```
Input: $\mathcal{G}(V, E)$
                                    $\triangleright$ perform tentative colouring on $\mathcal{G}$; round 0
#pragma omp parallel for
for all vertices $V_i \in \mathcal{G}$ do
    $\mathcal{C} \leftarrow$ {colours of all coloured vertices $V_j \in adj(V_i)$}
    $c(V_i) \leftarrow$ {smallest colour $\notin \mathcal{C}$}
end for
#pragma omp barrier
$\mathcal{U}^0 \leftarrow V$                        $\triangleright$ mark all vertices for inspection
$i \leftarrow 1$                                        $\triangleright$ round counter
while $\mathcal{U}^{i-1} \neq \emptyset$ do      $\triangleright$ $\exists$ vertices (re-)coloured in the previous round
    $\mathcal{L} \leftarrow \emptyset$      $\triangleright$ global list of vertices found to be defectively coloured
    #pragma omp parallel for
    for all vertices $V_i \in \mathcal{U}^{i-1}$ do        $\triangleright$ i.e. (re-)coloured in round $i-1$
                        $\triangleright$ if they are (still) defective, re-colour them immediately
        if $\exists V_j \in adj(V_i) : c(V_i) == c(V_j)$ and $id(V_i) < id(V_j)$ then
            $\mathcal{C} \leftarrow$ {colours of all coloured vertices $V_j \in adj(V_i)$}
            $c(V_i) \leftarrow$ {smallest colour $\notin \mathcal{C}$}
            $\mathcal{L} \leftarrow \mathcal{L} \cup V_i$            $\triangleright$ $V_i$ has been re-coloured in this round
        end if
    end for
    #pragma omp barrier
    $\mathcal{U}_i \leftarrow \mathcal{L}$             $\triangleright$ Set of vertices to be inspected in the next round
    $i \leftarrow i + 1$                                 $\triangleright$ proceed to next round
end while
```

formance and scalability on the kind of graphs we are mostly interested in, whereas RMAT graphs were used so that we are in line with the experimental methodology used in Çatalyürek's publication; the authors state that those RMAT graphs "are designed to represent instances posing varying levels of difficulty for the performance of multithreaded colouring algorithms".

For the 2D case we have used the adapted 2D mesh presented later in Section 6.5, named `bench_2d`, which consists of $\approx 250k$ vertices. This case gives a direct insight into how our improved method performs within PRAgMaTIc. We also evaluate performance using two 3D meshes, taken from the University of Florida Sparse Matrix Collection [30]. `bmw3_2` is a mesh modelling a BMW Series 3 car consisting of $\approx 227k$ vertices, whereas `pwtk` represents a pressurised wind tunnel and consists of $\approx 218k$ vertices.

Finally, we have generated three $16M$-vertex, $128M$-edge RMAT graphs in accordance with [20]. The RMAT algorithm generates a graph $\mathcal{G}(V, E)$ as follows. The adjacency matrix of $\mathcal{G}$, initially empty, is divided iteratively into four quadrants. Edges are distributed within the quadrants with specific probabilities *(a, b, c, d)*, which sum up to 1. For every edge, the algorithm places it in one of the four quadrants with probabilities *(a, b, c, d)*. The chosen quadrant is then subdivided into four new sub-quadrants and this process is repeated until we land onto a single cell, so an edge is created connecting the corresponding vertices.

The three RMAT graphs we use for our benchmarks are called `RMAT-ER` (Erdős-Rényi), `RMAT-G` (Good) and `RMAT-B` (Bad). They are generated using the following sets of probabilities:

- RMAT-ER: $(0.25, 0.25, 0.25, 0.25)$
- RMAT-G: $(0.45, 0.15, 0.15, 0.25)$
- RMAT-B: $(0.55, 0.15, 0.15, 0.15)$

Vertex indices are shuffled randomly so as to reduce the benefits of data locality and large caches. For more information on the characteristics of those graphs the reader is referred to the original publication by Çatalyürek *et al.* [20].

The experiments were run on two systems: a dual-socket Intel®Xeon® E5-2650 system (Sandy Bridge, 2.00GHz, 8 physical cores per socket, 2-way hyper-threading per core, 32 threads in total) running Red Hat®Enterprise Linux® Server release 6.4 (Santiago) and an Intel®Xeon Phi™ 5110P board (1.053GHz, 60 physical cores, 4-way hyper-threading per core, 240 threads in total). Both versions of the code (intel64 and mic) were compiled with Intel®Composer XE 2013 SP1 and with the compiler flags `-O3 -xAVX`. The benchmarks were run using Intel®'s thread-core affinity support.

In our experiments we compare the new improved method, henceforth referred to as "Rokos *et al.*", with the one by Çatalyürek *et al.*, since the latter is the up-to-now state-of-the-art greedy colouring algorithm. The rest of the algorithms are of little interest nowadays and the corresponding results have been omitted. Jones-Plassmann has been surpassed by newer innovations (*i.e.* optimistic algorithms), whereas its multi-hash version was found to produce very bad colourings, using on average 5-10 times more colours than the rest of the algorithms presented in this chapter. Finally, Gebremedhin-Manne is just the predecessor to Çatalyürek *et al.*, with the latter being an optimised, better performing version of the former.

Figure 5.2 shows the total execution time on Intel®Xeon® for both algorithms to colour the three meshes `bench_2d`, `bmw3_2` and `pwtk`, whereas Figure 5.3 shows the execution time for the three RMAT graphs. Figures 5.4 and 5.5 show the corresponding achievable speedup over the single-threaded case as the number of thread increases. Figures 5.6, 5.7, 5.8 and 5.9 present the same benchmarks on Intel®Xeon Phi™. As can be seen, Rokos *et al.* performs faster than Çatalyürek *et al.* for every test graph on both platforms, while scaling significantly better as the number of threads increases, especially on Intel®Xeon Phi™.

Figures 5.10 and 5.11 show the relative speedup of Rokos over Çatalyürek for all test graphs on Intel®Xeon® and Intel®Xeon Phi™, respectively. The gap between the two algorithms widens with the number of threads, reaching a maximum value of 50% on Intel®Xeon Phi™ for `RMAT-B`.

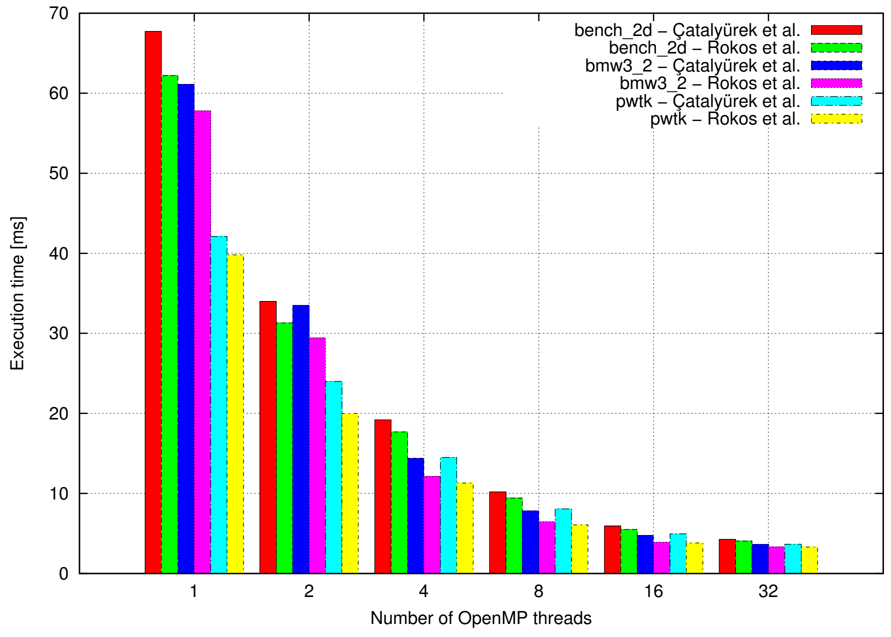


Figure 5.2: Execution time on Intel®Xeon® E5-2650 (meshes).

Figure 5.12 depicts the total number of conflicts detected throughout the whole execution of both algorithms for the three meshes on Intel®Xeon®

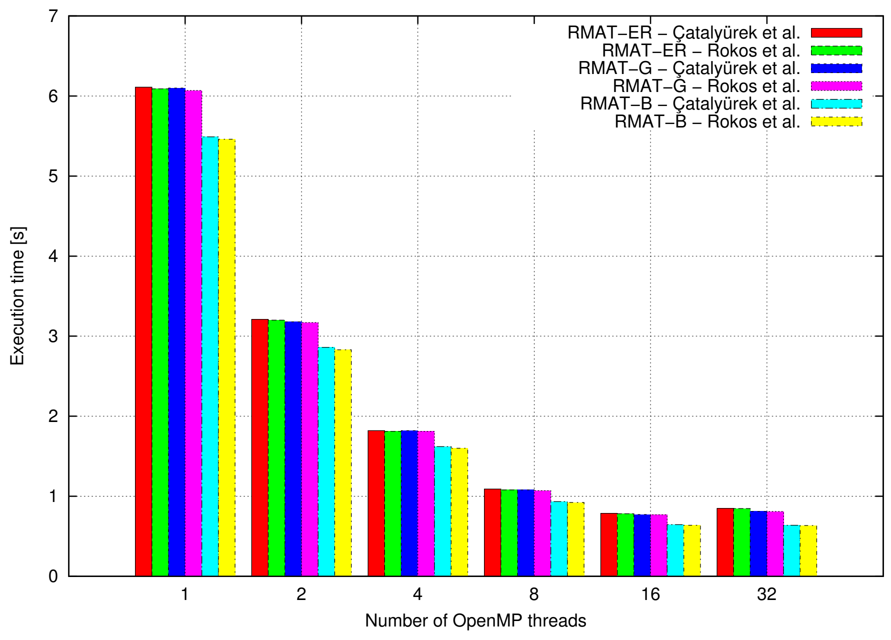


Figure 5.3: Execution time on Intel®Xeon® E5-2650 (RMAT).

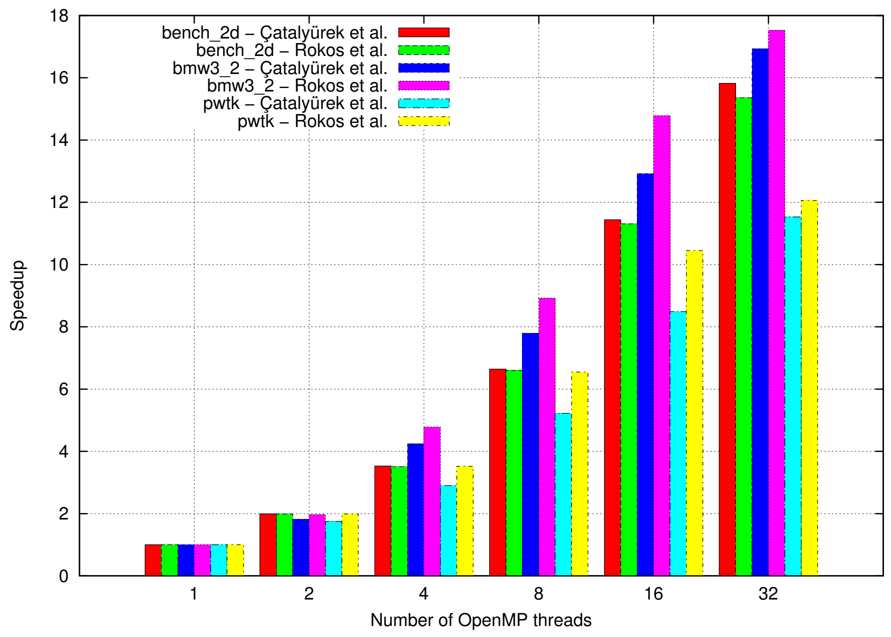


Figure 5.4: Speedup over single-threaded execution on Intel®Xeon® E5-2650 (meshes).

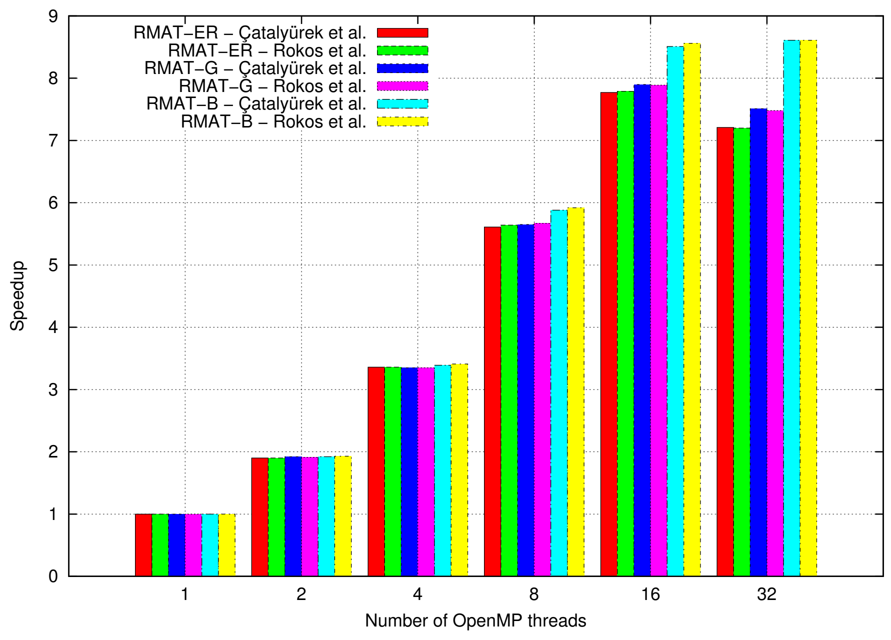


Figure 5.5: Speedup over single-threaded execution on Intel®Xeon® E5-2650 (RMAT).

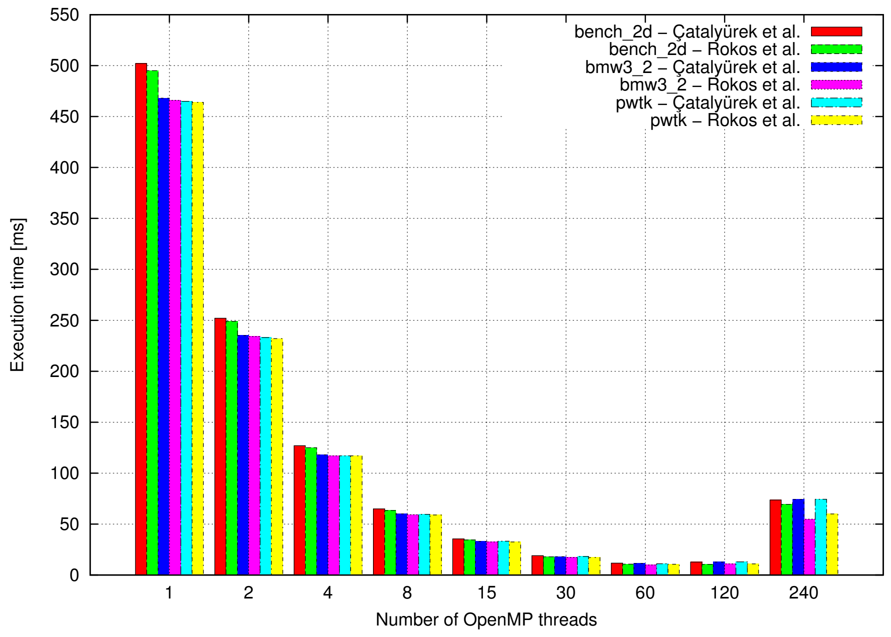


Figure 5.6: Execution time on Intel®Xeon Phi™ 5110P (meshes).

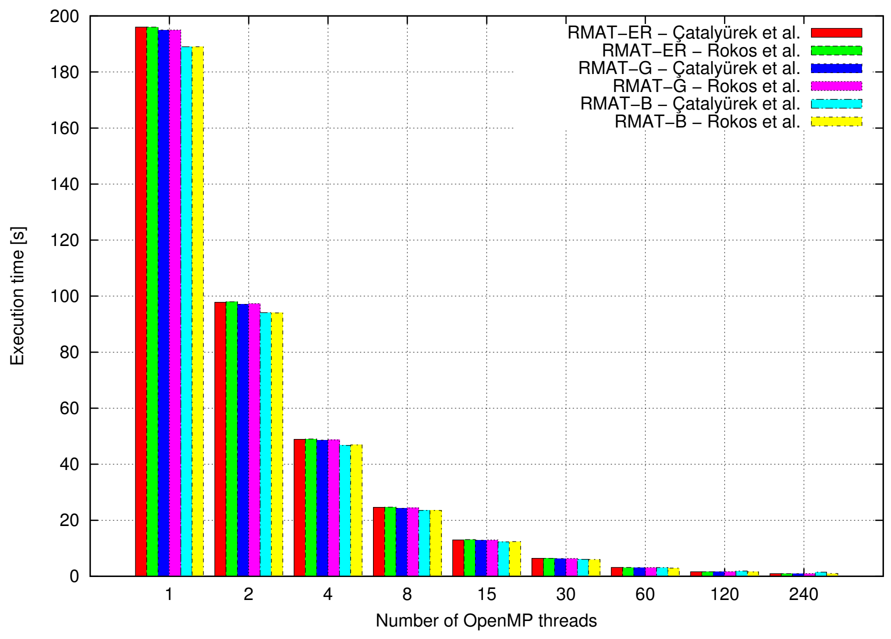


Figure 5.7: Execution time on Intel®Xeon Phi™ 5110P (RMAT).

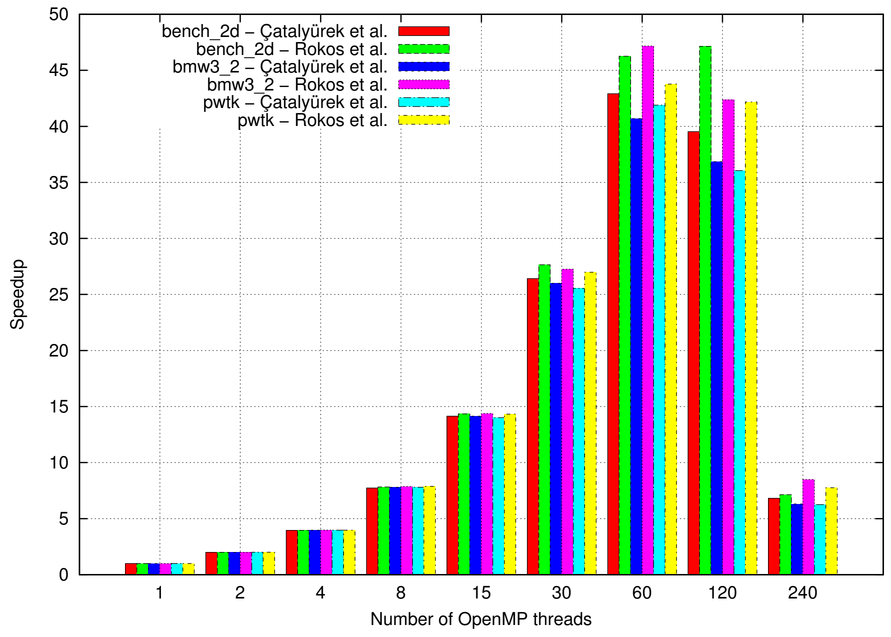


Figure 5.8: Speedup over single-threaded execution on Intel®Xeon Phi™ 5110P (meshes).

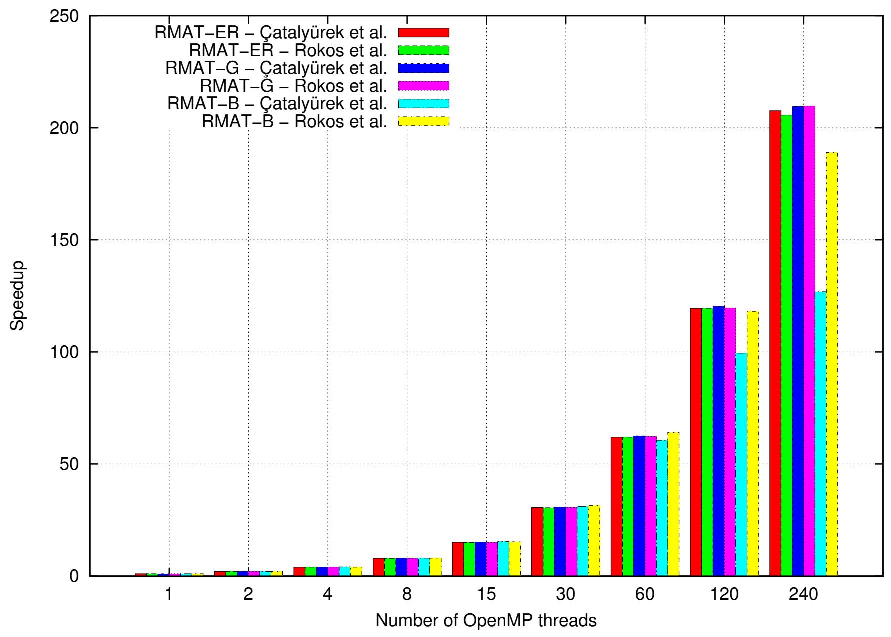


Figure 5.9: Speedup over single-threaded execution on Intel®Xeon Phi™ 5110P (RMAT).

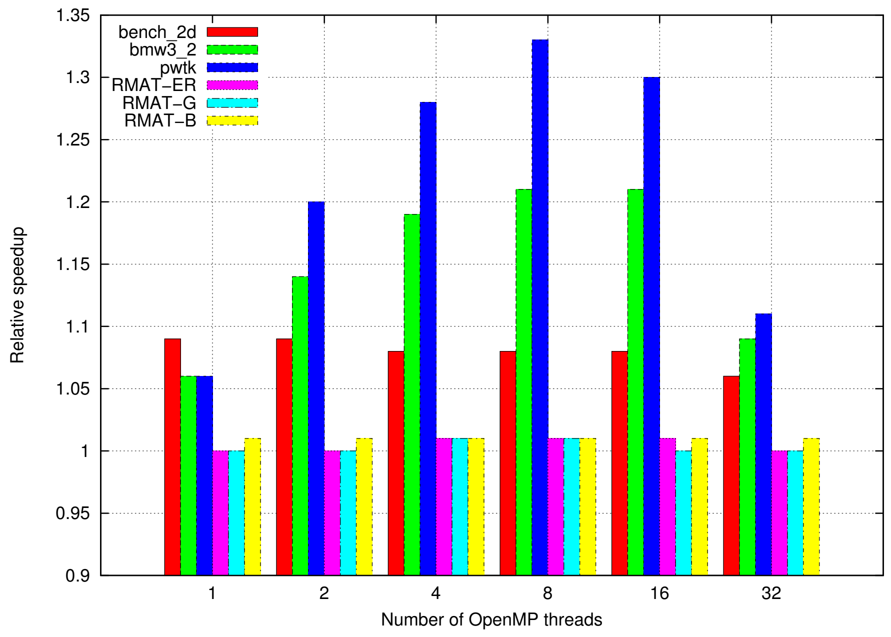


Figure 5.10: Speedup of Rokos over Çatalyürek on Intel®Xeon® E5-2650.

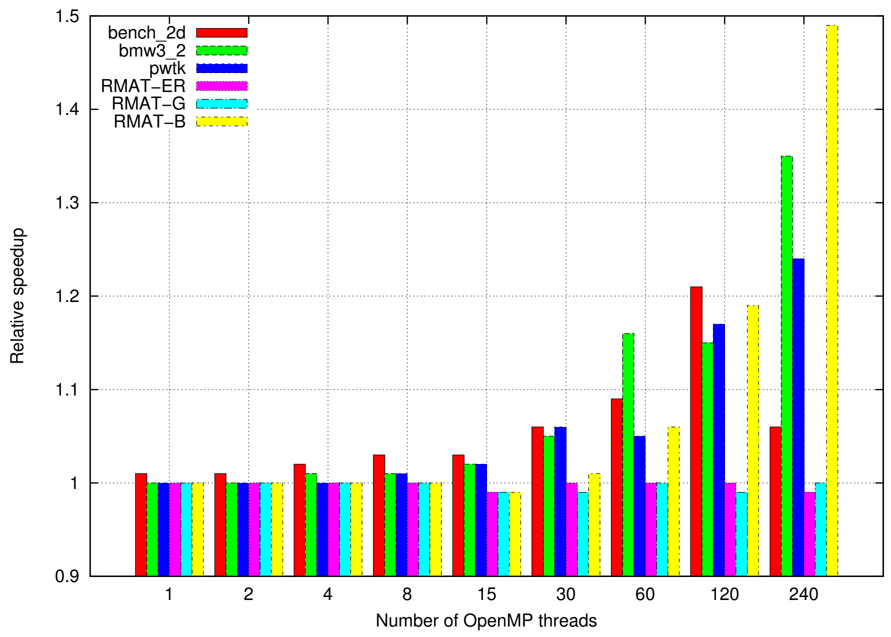


Figure 5.11: Speedup of Rokos over Çatalyürek on Intel®Xeon Phi™ 5110P.

and Figure 5.13 shows the corresponding measurements for the RMAT graphs. Similarly, Figures 5.14 and 5.15 present the same information on Intel®Xeon Phi™. When the number of threads is low, both algorithms produce more or less the same amount of conflicts. However, moving to higher levels of parallelism on Intel®Xeon Phi™ reveals that Rokos *et al.* results to significantly fewer defects in colouring for certain classes of graphs.

This observation can be explained as follows: In Çatalyürek *et al.*, when a thread detects a defect in colouring during the tentative-colouring phase, it appends the problematic vertex into a global worklist. Before entering the conflict-resolution phase, all participating threads synchronise, which means that they enter that phase and start resolving conflicts at the very same time. Therefore, it is highly possible that two adjacent vertices with conflicting colours will be processed by two threads simultaneously, which leads once again to defective colourings. In our improved algorithm, on the other hand, a conflict is resolved as soon as it is discovered by a thread. The likelihood that another thread is visiting and recolouring a neighbouring vertex at the same time is certainly lower than in Çatalyürek *et al.*.

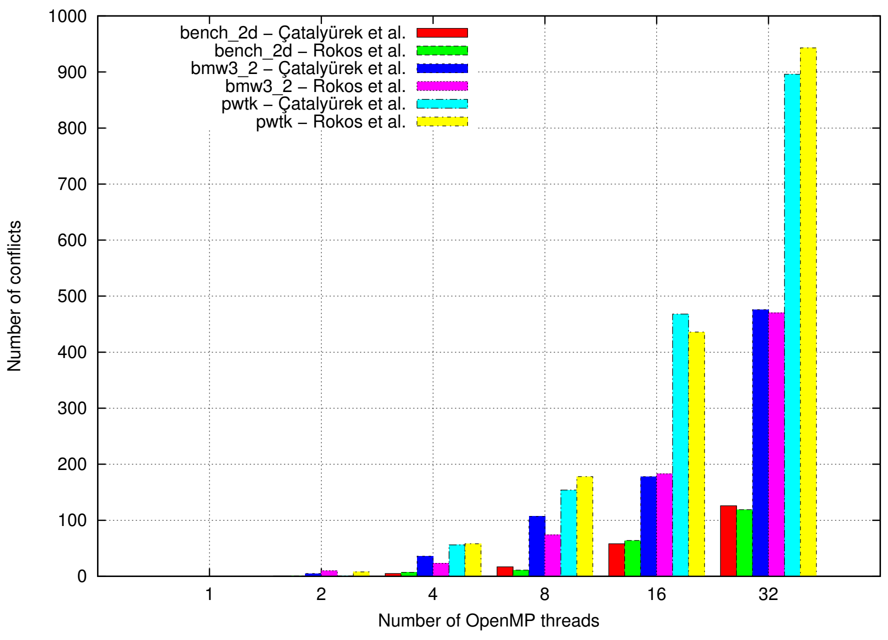


Figure 5.12: Number of conflicts on Intel®Xeon® E5-2650 (meshes).

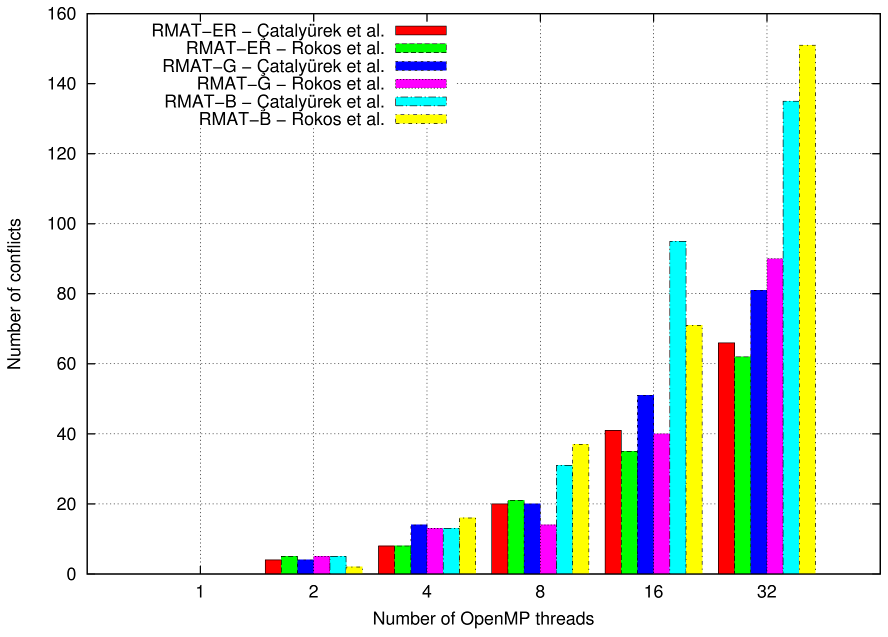


Figure 5.13: Number of conflicts on Intel®Xeon® E5-2650 (RMAT).

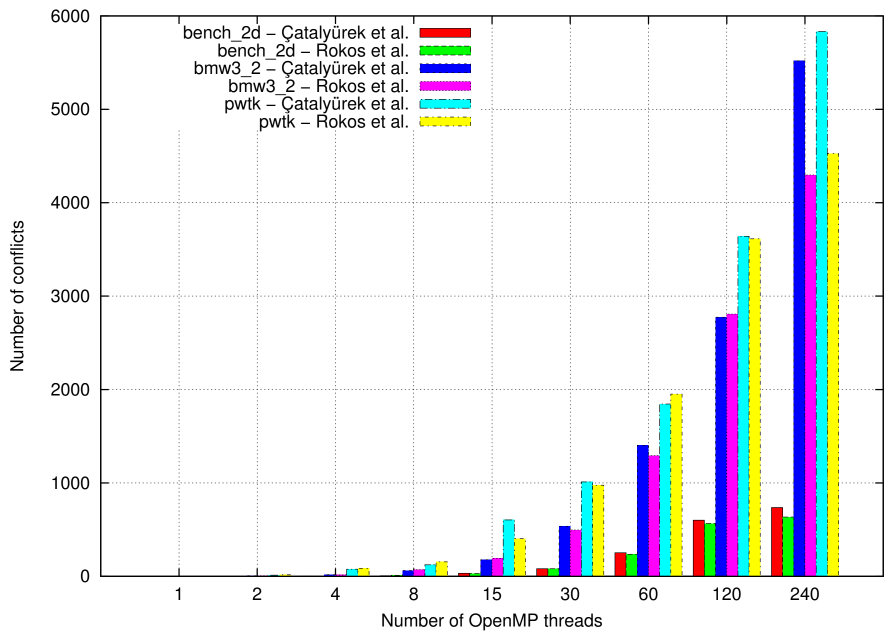


Figure 5.14: Number of conflicts on Intel®Xeon Phi™ 5110P (meshes).

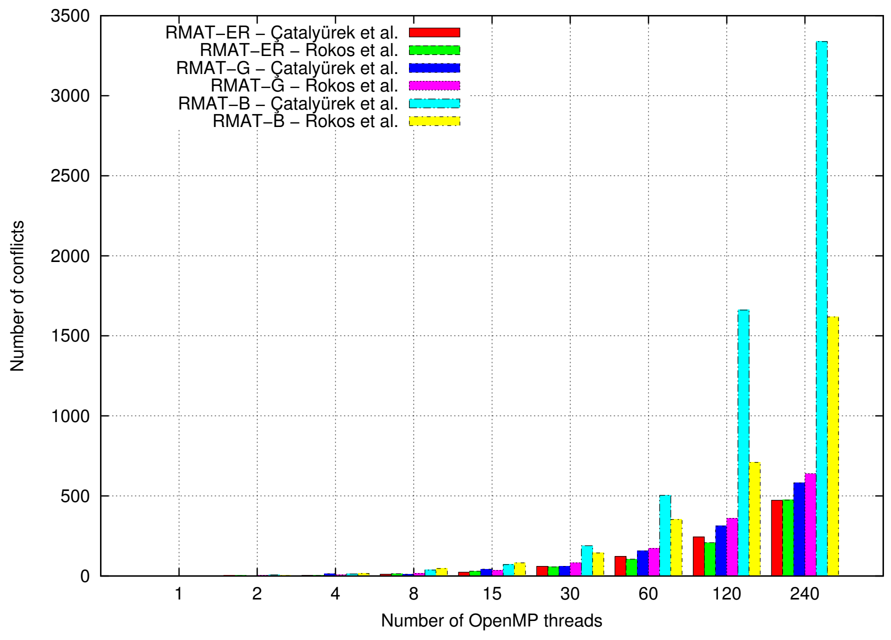


Figure 5.15: Number of conflicts on Intel®Xeon Phi™ 5110P (RMAT).

The reduced amount of conflicts also results in fewer iterations of the algorithm, as can be seen in Figures 5.16 and 5.17 for Intel®Xeon® and Figures 5.18 and 5.19 for Intel®Xeon Phi™. Taking into account that every iteration of the while loop in Rokos *et al.* involves only one thread synchronisation point as opposed to two in Çatalyürek *et al.*, it is only expected that our new algorithm ultimately outperforms its predecessor. A nice property is that both algorithms produce colourings using the same number of colours, *i.e.* quality of colouring is not compromised by the higher execution speed.

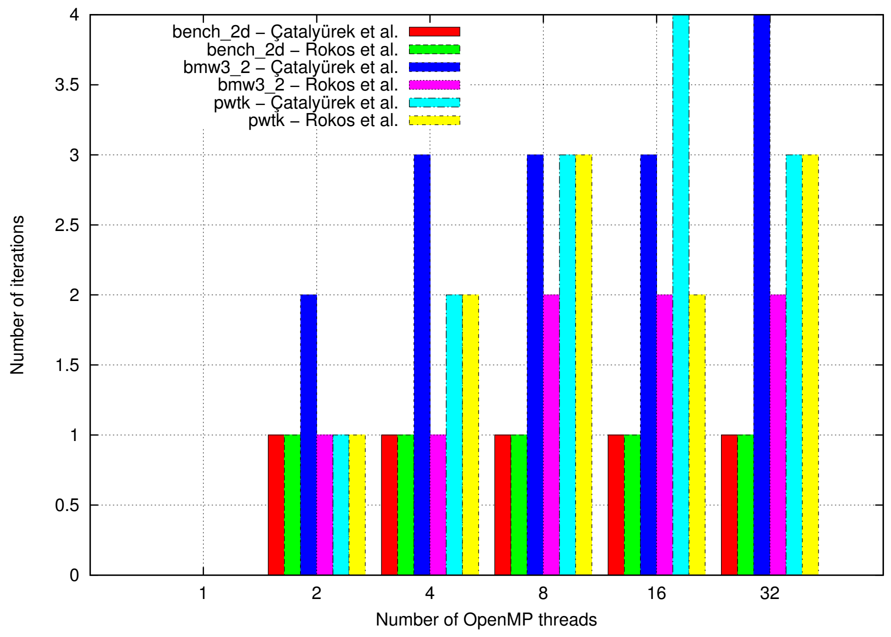


Figure 5.16: Number of iterations on Intel®Xeon® E5-2650 (meshes).

5.9 SIMT restrictions

Trying to run the optimistic colouring algorithms on CUDA revealed a potential weakness. Both the algorithm by Çatalyürek *et al.* and our improved version never ran to completion; instead, threads spun forever in an infinite loop. This is due to the nature of SIMT-style multi-threading, in which the lockstep warp execution results in ties never being broken.

An example of why these algorithms result in infinite loops in SITM-style

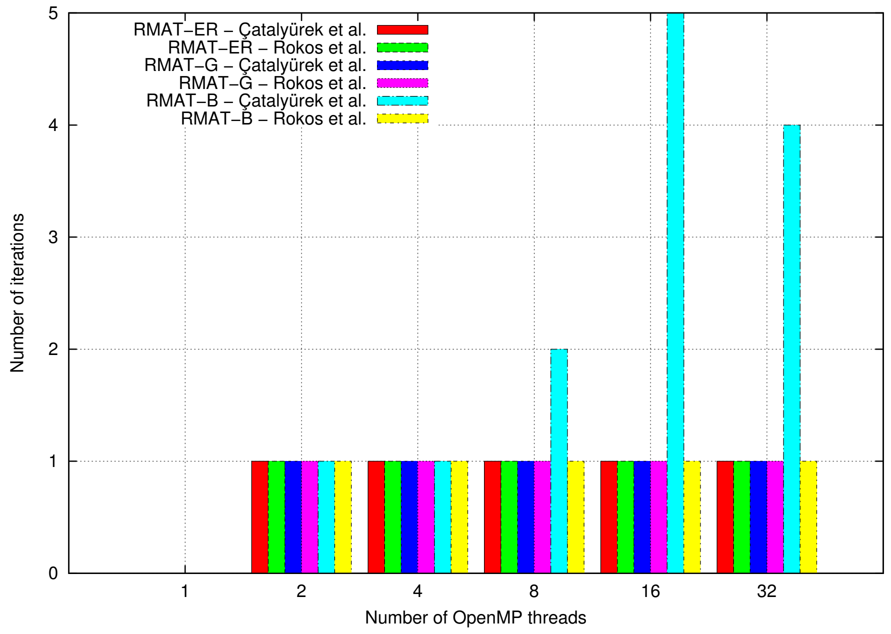


Figure 5.17: Number of iterations on Intel®Xeon® E5-2650 (RMAT).

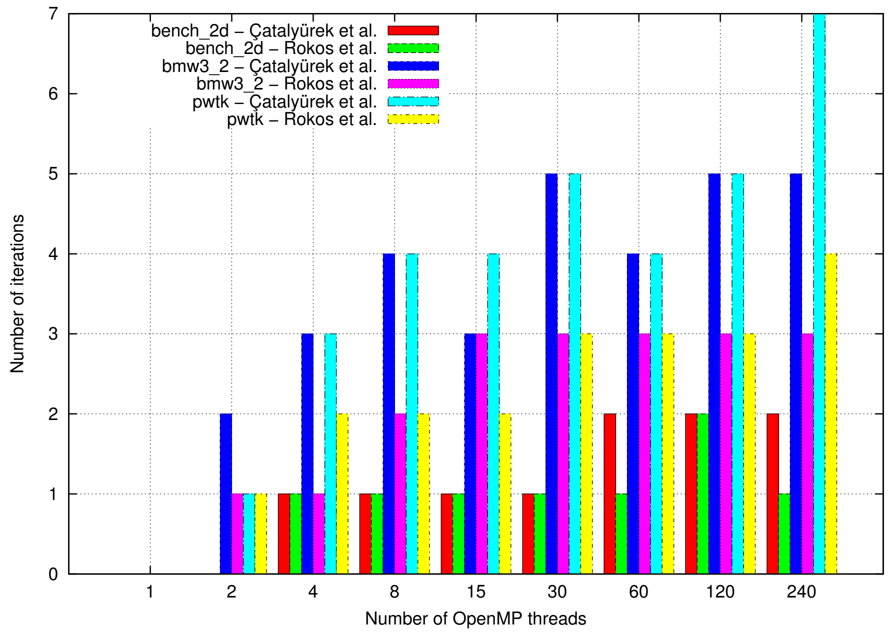


Figure 5.18: Number of iterations on Intel®Xeon Phi$^{\text{TM}}$ 5110P (meshes).

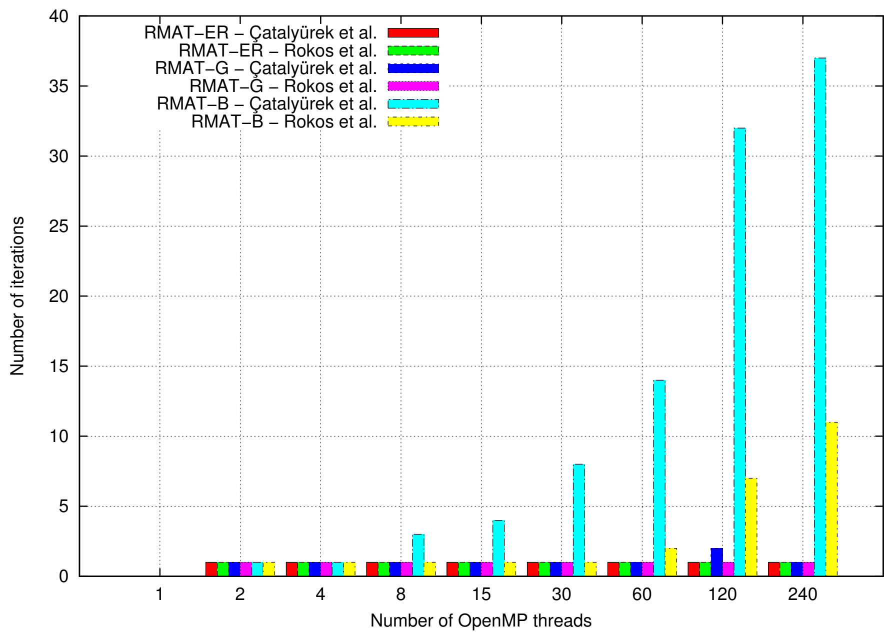


Figure 5.19: Number of iterations on Intel®Xeon Phi™ 5110P (RMAT).

parallelism can be seen in Figure 5.20, where we have a simple two-vertex graph and two threads, each processing one vertex. Colour ordering is the same as in Figure 5.1, *i.e.* {red, green, yellow, blue, ...}. At the beginning (a), both vertices are uncoloured. Each thread sees that the adjacent vertex is uncoloured and decides that the smallest colour available for its own vertex is red. Both threads commit their decision at the exact same clock cycle, which results in the defective colouring shown in (b). In the next round, both threads detect the conflict by reading the neighbour's colour at the same clock cycle and try to resolve it. Both threads decide that the new smallest colour available is green and assign it to their vertices at the same clock cycle, resulting once again in defects (c). Next up, the conflict is detected, each thread finds out that the smallest colour available for its vertex is red, the colour is committed by both threads at the same clock cycle and the process goes on forever.

Theoretically, this scenario is possible for CPUs as well, although the probability is extremely low. Predictions are impossible to make, as convergence rate depends on too many factors (graph structure, data types used, processor speed, memory latency, memory bandwidth, number of threads,

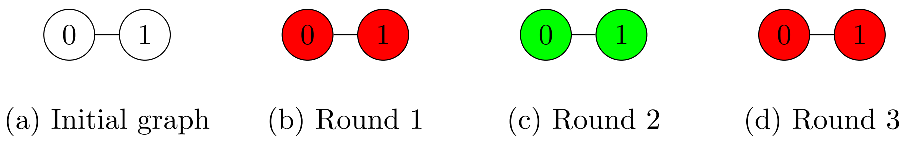


Figure 5.20: Example of an infinite loop in SITM-style parallelism when using one of the optimistic colouring algorithms.

interference from the operating system etc.). Nonetheless, we believe that there will always be some randomness and thread divergence on CPUs which guarantees convergence of the optimistic algorithms.

5.10 Conclusions

In this chapter we presented various parallel graph colouring algorithms and showed how we devised an improved version which outperforms its competitors, being up to 50% faster than the runner up for certain classes of graphs and scaling better on manycore architectures. The difference becomes more obvious as we move to graphs with higher-degree vertices (3D meshes, RMAT-B graph).

This observation also implies that our method (with the appropriate extensions) could be a far better option for 2-distance colourings of a graph $\mathcal{G}$, where $\mathcal{G}^2$, the 2nd power graph of $\mathcal{G}$, is considerably more densely connected. (The graph $\mathcal{G}^d$, the d^{th} power graph of $\mathcal{G}$, has the same vertex set as $\mathcal{G}$ and two vertices in $\mathcal{G}^d$ are connected by an edge if and only if the same vertices are within distance d in $\mathcal{G}$.)

Speed and scalability stem from two sources, (a) reduced amount of conflicts which also results in fewer iterations and (b) reduced thread synchronisation per iteration. Colouring quality remains at the same levels as in older parallel algorithms, which in turn are very close to the serial greedy algorithm, meaning that they produce near-optimal colourings for most classes of graphs.

6 Threaded Implementation of Mesh Adaptivity

In this chapter we present how the four adaptive algorithms described in Chapter 2, namely coarsening, refinement, swapping and smoothing, are parallelised and implemented in PRAgMaTIc using the irregular compute methodology analysed in Chapter 4.

6.1 Edge 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 $\mathcal{G}$ is the mesh itself.

In Section 4.4, Figure 4.2 demonstrated 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 [31, 71].

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 14 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 15; and the second

Algorithm 14 Edge collapse.

```
Allocate $dynamic\_vertex, worklist$.
#pragma omp parallel
    #pragma omp for schedule(static)
    for all vertices $V_i$ do $dynamic\_vertex[V_i] \leftarrow -2$
    end for
    repeat
        #pragma omp for schedule(guided)
        for all vertices $V_i$ do
            if $dynamic\_vertex[V_i] == -2$ then
                $dynamic\_vertex[V_i] \leftarrow$COARSEN_IDENTIFY($V_i$)
            end if
        end for
        if dynamic vertex count == 0 then break
        end if
        Colour active sub-mesh
        for all independent sets $\mathcal{I}_n$ do
            #pragma omp for schedule(guided)
            for all $V_i \in \mathcal{I}_n$ do
                if $V_i$ has been un-coloured then skip $V_i$
                end if
                                        ▷ mark all neighbours for re-evaluation
                for all vertices $V_j \in$ NNList[$V_i$] do
                    $dynamic\_vertex[V_j] \leftarrow -2$
                end for
                if colour of target $V_t$ clashes then un-colour $V_t$
                end if
                $dynamic\_vertex[V_i] \leftarrow -1$
                COARSEN_KERNEL($V_i$)
            end for
            Commit deferred operations.
        end for
    until true
```

phase actually applies the coarsening operation, see Algorithm 16. 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

Algorithm 15 coarsen_identify

```
procedure COARSEN_IDENTIFY($V_i$)
    $S_i \leftarrow$ the set of all edges connected to $V_i$
    $S^0 \leftarrow S_i$
    repeat
        $E_j \leftarrow$ 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
        end if
        $V_t \leftarrow$ the other vertex that bounds $E_j$
        evaluate collapse of $E_j$ with the collapse of $V_i$ onto $V_t$
        if ($\forall$ edges $\in S_i \leq L_{max}$) and($\nexists$ inverted elements) then
            return $V_t$
        else
            remove $E_j$ from $S^j$             ▷ $E_j$ is not a candidate for collapse
        end if
    until $S_i = \emptyset$
end procedure
```

two deleted elements from the element list.

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 colour what we call the active sub-mesh, i.e. the subset of all active vertices, and create independent sets $\mathcal{I}_n$. Working with independent sets not only ensures safe parallel execution, but also enforces the *every other vertex* rule. For every active vertex $V_r \in \mathcal{I}_n$ which is about to collapse, the

Algorithm 16 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
    end for
    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
    end for
    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
    end for
    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
    end for
    NNList[$V_i$].clear()
    NEList[$V_i$].clear()
end procedure
```

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. Additionally, removal of V_r may introduce defects in colouring between the target vertex V_t (the one V_r collapses onto) and all other vertices V_a of the local neighbourhood. In this case, V_t is un-coloured and will be skipped when processing the independent set it belongs to. V_t remains marked as active, so will be processed in a subsequent iteration of coarsening.

Algorithm 16 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 set, all deferred operations are committed.

6.2 Element 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 Section 4.5, 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 Algorithm 17 and Algorithm 18, 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 $\overline{V_iV_j}$ is considered to be different from edge $\overline{V_jV_i}$. 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 $index = NNodes$ 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

Algorithm 17 Edge-refinement.

```
Global worklist of split edges $\mathcal{W}$, refined_edges_per_element[NElements]
#pragma omp parallel
    $private : split\_cnt \leftarrow 0, newCoords, newMetric, newVertices$
    #pragma omp for schedule(guided)
    for all vertices $V_i$ do
        for all vertices $V_j$ adjacent to $V_i$, $ID(V_i) < ID(V_j)$ do
            if $length\ of\ edge\ \overline{V_iV_j} > L_{max}$ then
                $V_n \leftarrow$ new vertex of split edge $\overline{V_iV_nV_j}$
                Append new coordinates to $newCoords$
                Append interpolated metric to $newMetric$
                Append split edge to $newVertices$
                $split\_cnt \leftarrow split\_cnt + 1$
            end if
        end for
    end for
    #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
        $e_i = \overline{V_iV_nV_j}$; increment ID of $V_n$ by $index$
    end for
    Copy $newVertices$ into $\mathcal{W}$
    #pragma omp barrier
    #pragma omp parallel for schedule(guided)
    for all Edges $e_i \in \mathcal{W}$ 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[$E_i$].
        end for
    end for
```

into a global worklist. For each split edge $\overline{V_iV_j}$, 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 elements 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.

Algorithm 18 Element refinement phase

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

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.

This parallel refinement algorithm has the advantage of not requiring any mesh colouring and having low synchronisation overhead as compared with Freitag's task graph approach.

6.3 Edge 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 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* is a vector of sets (`std::vector< std::set<int> >`).

The algorithm begins by marking all edges. It then enters a loop which is terminated when no marked edges remain. The active sub-mesh is coloured and independent sets $\mathcal{I}_n$ are calculated. A vertex is considered active if at least one of its outbound edges is marked. Following that, threads process in parallel all active vertices of an independent set $\mathcal{I}_n$. 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 Figure 2.10 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 $V_0V_1V_2V_3$) have to be marked for processing. This is how propagation is implemented in swapping.

One last difference between swapping and coarsening is that an independent set $\mathcal{I}_n$ 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 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.

6.4 Quality-Constrained Vertex Smoothing

Algorithm 19 illustrates the colouring based algorithm for mesh smoothing. In this algorithm the operation task 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 kernels in Section 2.6.4.

Algorithm 19 Thread-parallel mesh smoothing

```
repeat
    $relocate\_count \leftarrow 0$
    for $colour = 1 \rightarrow k$ do
        #pragma omp for schedule(guided)
        for all $i \in \mathcal{V}^c$ do
                                        ▷ $move\_success$ is true if vertex was relocated,
            $move\_success \leftarrow smooth\_kernel(i)$        ▷ false otherwise.
            if $move\_success$ then
                $relocate\_count \leftarrow relocate\_count + 1$
            end if
        end for
    end for
until $(n \geq max\_iteration)$ or $(relocate\_count = 0)$
```

6.5 Experimental 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:

$$\psi(x, y, t) = 0.1 \sin\left(50x + \frac{2\pi t}{T}\right) + \arctan\left(-\frac{0.1}{2x - \sin\left(5y + \frac{2\pi t}{T}\right)}\right) \qquad (6.1)$$

where T is the period. An example of the field at $t = 0$ is shown in Figure 6.1. 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. Figure 6.2 shows the adapted mesh in which every element is coloured depending on its quality, as can be seen in the legend. A magnified region around the lower sinusoidal front demonstrating the variation of element quality in higher detail can be seen in Figure 6.3.

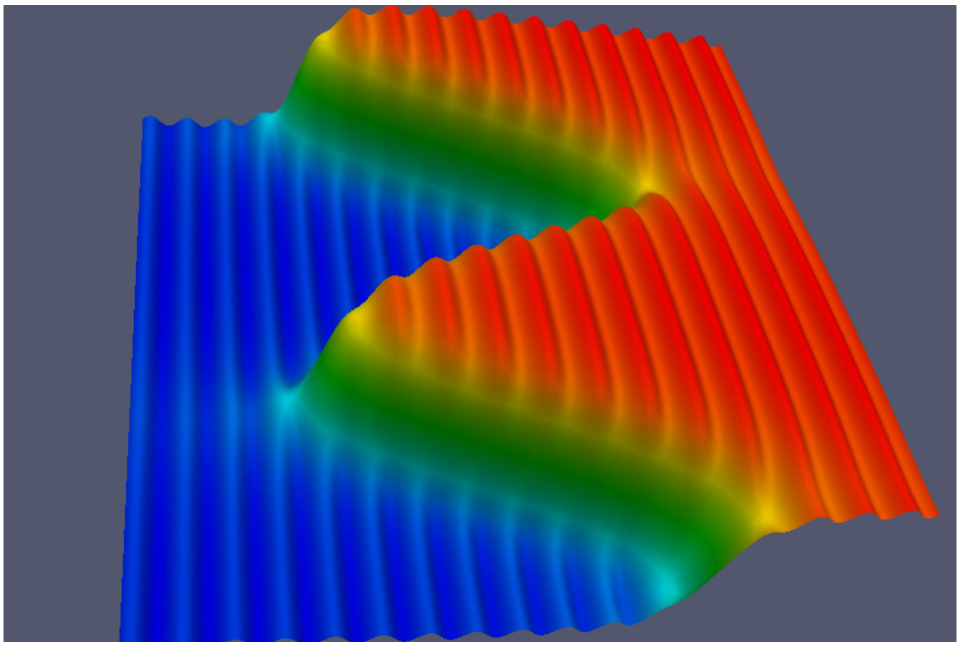


Figure 6.1: Benchmark solution field for time step t_0.

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 aggregated time when reporting performance results. To calculate the metric we used the $L^{p=2}$-norm as described by [22]. 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 (Figure 6.4) showing the quality of all elements aggregated over all time steps. We can see that the vast majority of the elements are of very high quality. The lowest quality element had a quality of 0.51, and in total only 40 thousand elements out of 26 million have a quality of less than 0.6.

The benchmarks were run on a dual-socket Intel®Xeon® E5-2650 system

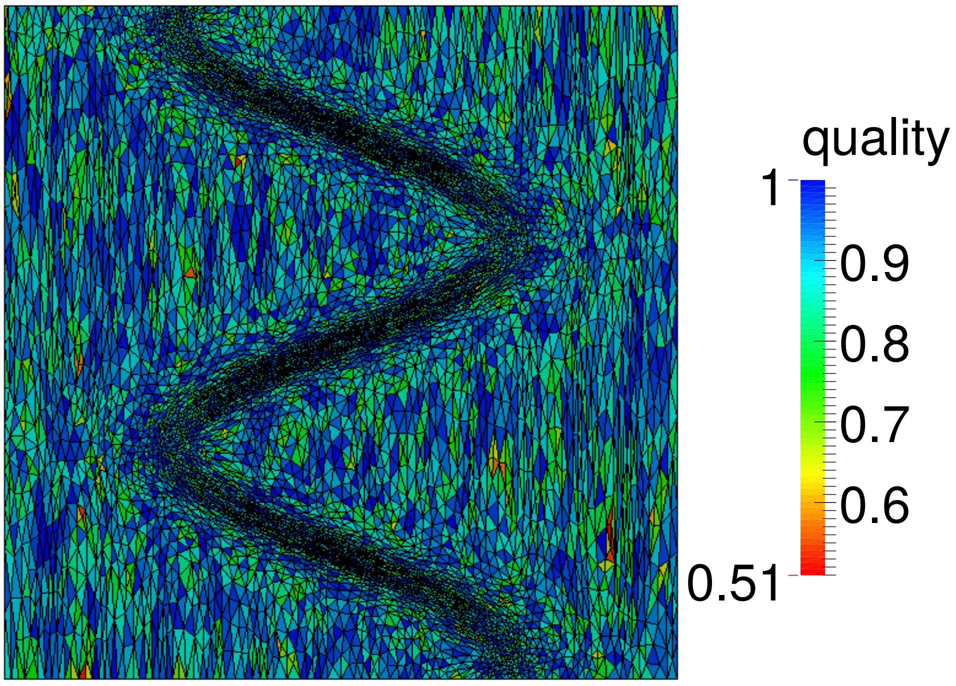


Figure 6.2: Quality of adapted mesh for time step t_0. As can be seen in the legend, red elements are low-quality ones whereas blue elements are of higher-quality (close to ideal).

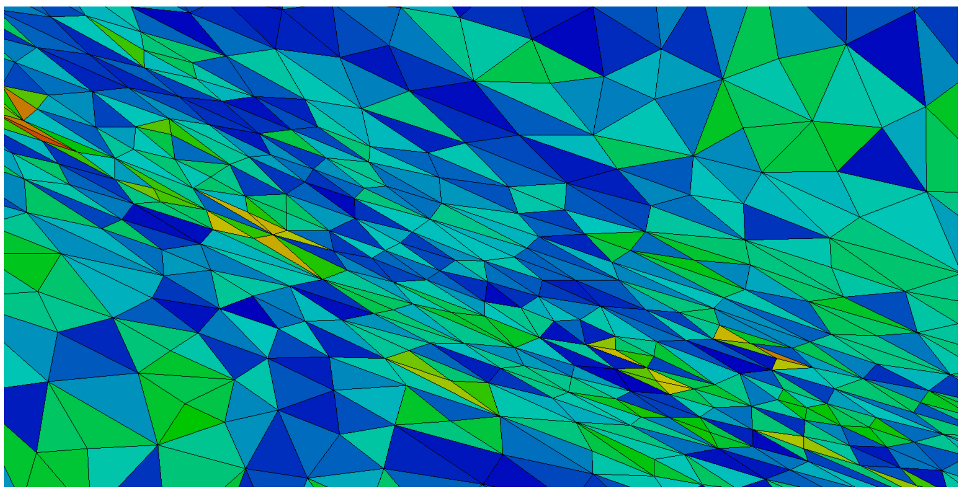


Figure 6.3: Magnification of mesh from Figure 6.2 around the lower region of the sinusoidal front, demonstrating the variation of element quality in this highly anisotropic area.

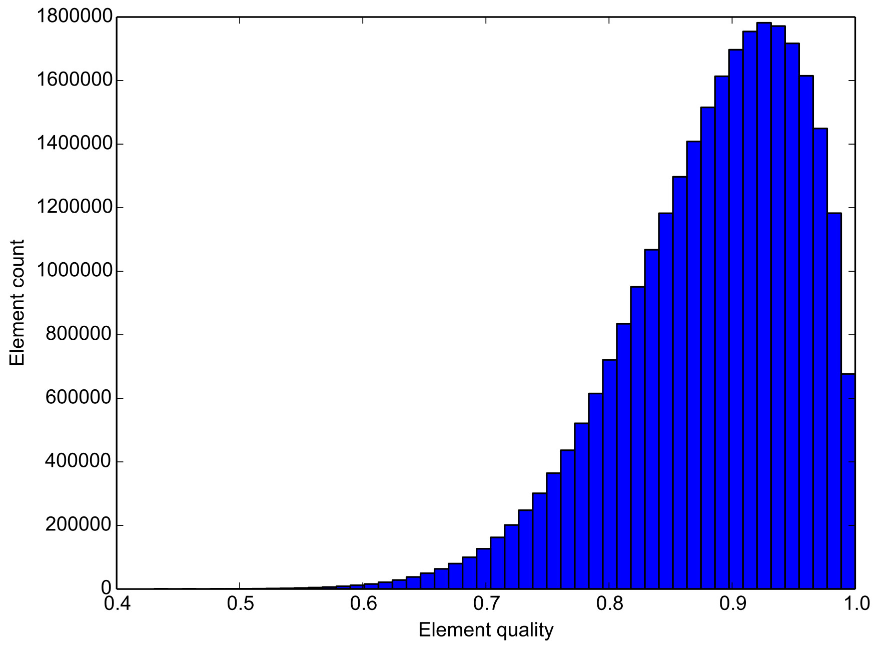


Figure 6.4: Histogram of all element quality aggregated over all time steps.

(Sandy Bridge, 2.00GHz, 8 cores per socket, 2-way hyper-threading per core). The code was compiled using the Intel compiler suite, version 14.0.1 and with the compiler flags `-O3 -xAVX`. In all cases, thread-core affinity was used.

Figures 6.5, 6.6 and 6.7 show the wall time, speedup and efficiency of each phase of mesh adaptation and for the overall procedure (sum of the four adaptive algorithms and mesh defragmentation, a necessary processing step to deal with gaps in data structures as a result of element/node deletion during coarsening). Simulations using between 1 and 8 threads are run on a single socket with every thread running on its own exclusive core. The 16-thread simulation runs across two CPU sockets (8 cores per socket, 1 thread per core), thereby incurring NUMA overheads. Finally, the simulation marked as 32(HT) on the diagrams is the NUMA simulation with hyper-threading enabled (2 threads per core), using all 32 hardware threads available in the system. From the results we can see that all operations achieve good scaling, including the 16-core NUMA case.

It is notable that PRAgMaTIc needs $\approx 1.5s$ for each time step when using

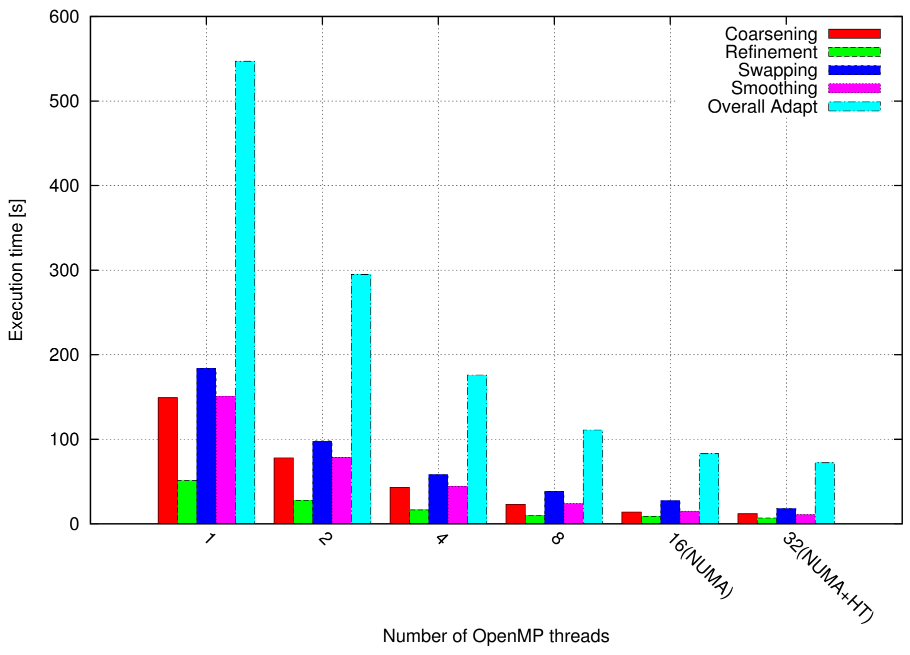


Figure 6.5: Wall time for each phase of mesh adaptation.

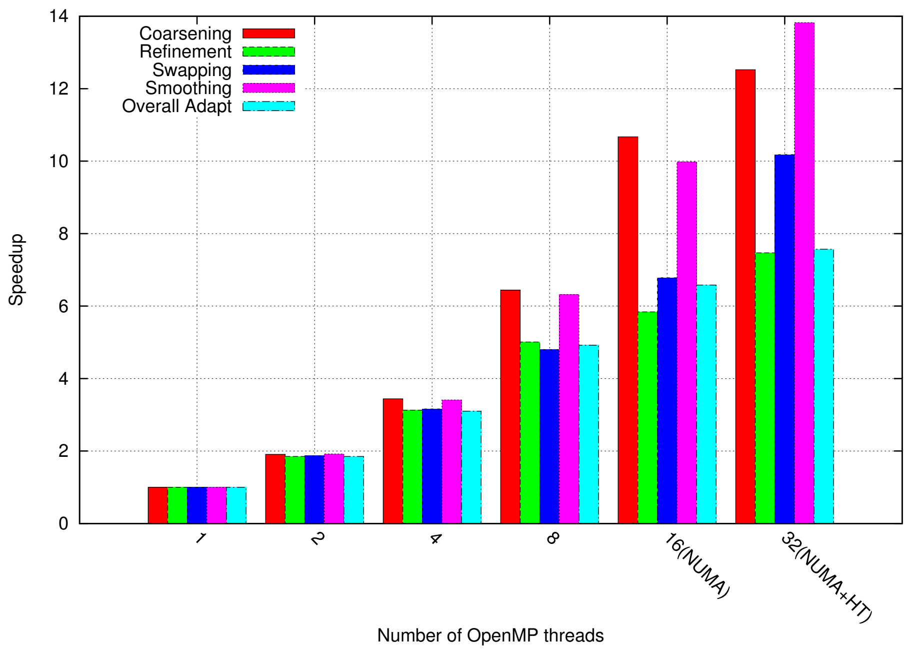


Figure 6.6: Speedup profile for each phase of mesh adaptation.

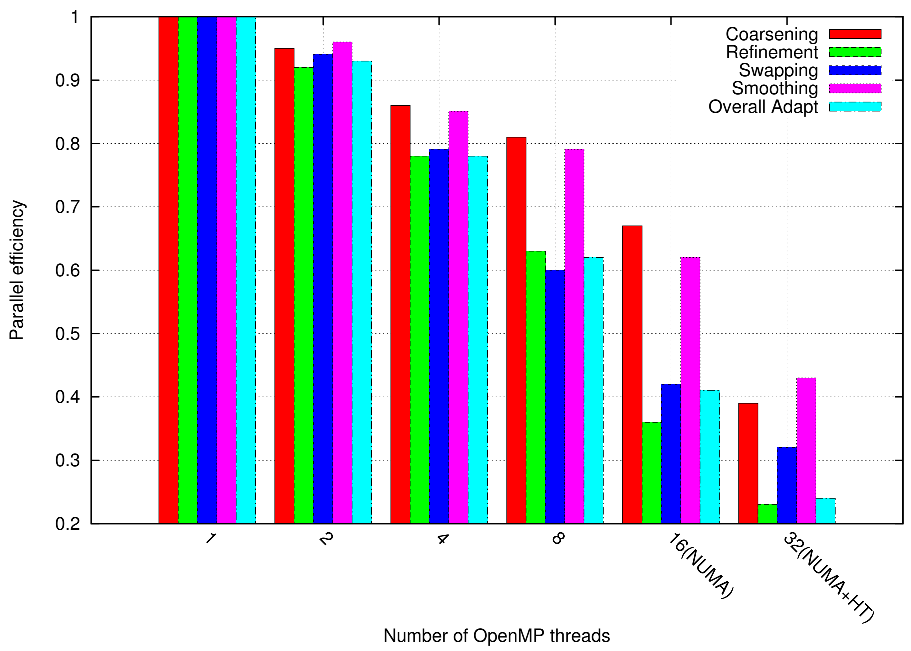


Figure 6.7: Parallel efficiency profile for each phase of mesh adaptation.

all available hardware resources of the dual-socket Intel®Xeon® machine. This is relatively low compared to typical solution times for CFD problems, for example [citation pending].

Figure 6.8 shows a comparison of the execution time per adaptive kernel between the aforementioned Intel®Xeon® system and an Intel®Xeon Phi™ 7120P board (1.238GHz, 61 physical cores, 4 hyperthreads per core, 244 threads in total). Code for Intel®Xeon Phi™ was compiled with the same compiler and optimisation flags as the Sandy-Bridge version. The figure shows results for 61 threads (61 physical cores, no hyperthreading) and 122 threads (61 physical cores, 2 hyperthreads per core). We are presenting those particular configurations because, whereas performance is improved as we increase the number of threads on Intel®Xeon Phi™ up to 61 (results for fewer threads have been omitted), there is a slowdown for the swapping kernel when using 122 threads. Additionally, we observed a tremendous slowdown for all kernels when using 244 threads; we figured out that OpenMP's guided scheduling was the cause, which is unrelated to the parallel implementation of the adaptive algorithms, so the results for this case have been omitted and will be discussed in Chapter 7.

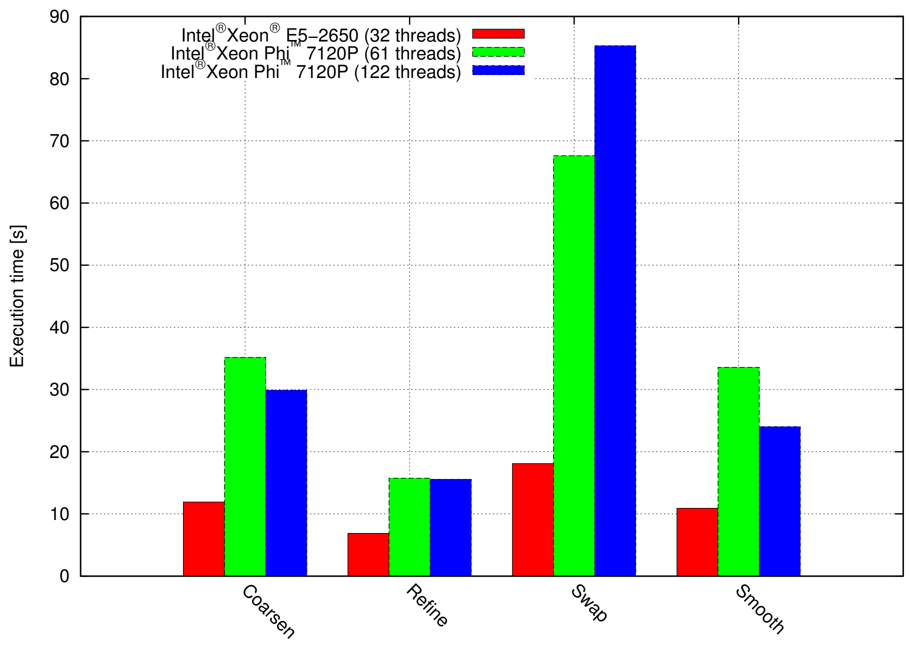


Figure 6.8: Comparison of execution time between Intel®Xeon® and Intel®Xeon Phi™.

The comparison of execution times between the two platforms reveals the dominant factors hindering performance in PRAgMaTIc. Under ideal circumstances, *i.e.* using perfectly optimised codes, a 61-core Intel®Xeon Phi™ would be expected to come close and/or match or even surpass a 16-core Intel®Xeon®, even if the latter is running at double the clock rate and has been built upon a significantly more sophisticated architectural design (out-of-order execution, branch prediction etc.). In [58] it is argued that "[t]he potential [of Intel®Xeon Phi™ compared to Intel®Xeon® in terms of peak performance] is higher, but so is the parallelism needed to get there". It is clear from Figure 6.8 that this is not the case here. Intel®Xeon Phi™ is far behind Intel®Xeon® for all algorithms and this suggests two major scalability bottlenecks: bandwidth saturation and thread synchronisation.

Refinement, which involves very little computation and is mainly dominated by memory transfers, is the kernel scaling the worst on both platforms, showing virtually no speedup beyond 61 threads on Intel®Xeon Phi™. Additionally, the fact that swapping runs slower with 122 threads is indicative of the amount of thread synchronisation. Swapping is the kernel with the highest barrier count. Given the bandwidth saturation and bad data locality, increasing the number of threads only makes things worse, as synchronisation overhead is increased without any opportunity to offset it by hiding memory access latency; on the contrary, having more threads per core causes additional demand for bandwidth which is not available any more. All these observations are in line with profiling results we got from hotspot and concurrency analysis in the Intel®VTune™Amplifier XE performance profiler.

Coarsening and smoothing, on the other hand, scale much better. Both kernels are less bandwidth-hungry compared to refinement and require much less thread synchronisation than swapping, while involving a fair amount of floating-point arithmetic (especially smoothing). It is only expected that those two kernels scale the best, still showing speedup when turning on hyperthreading on Intel®Xeon Phi™.

6.6 Conclusions

In this chapter we reviewed and evaluated the threaded implementation of the four adaptive algorithms described in Chapter 2 using the irregular compute methodology from Chapter 4. The techniques for irregular data we developed proved to be scalable and helped us build PRAgMaTIc, a demanding real-life irregular application which achieves good parallel efficiency even in NUMA configurations.

Summarising, the dominant factors limiting scalability are:

(a) The number of thread synchronisations (there are many unavoidable barriers in each adaptive algorithm).

(b) Memory access latency due to bad data locality, a common problem in applications with irregular data; the fact that enabling hyperthreading boosts PRAgMaTIc's performance indicates that our implementation is not compute bound and can be sped up by hiding/improving memory access latency.

(c) Load-imbalances between threads; 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. This observation motivated us to go after a better for-loop scheduling strategy based on work-stealing principles (here, OpenMP's *guided* scheduling was used for for-loops, which proved to be adequate for benchmarks on Intel®Xeon® but incurred tremendous overhead on Intel®Xeon Phi™). This experimental work is presented in the next chapter.

The fact that the parallel efficiencies of mesh refinement, coarsening, swapping and smoothing are comparable (up to a point) 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.

7 An Interrupt-Driven Work-Sharing For-Loop Scheduler

For-loops are a key component of most scientific applications. Running loops in parallel is a key component in accelerating a program and making the most of modern, massively multi-threaded systems. However, parallelisation does not come for free. Achieving good load balance usually involves some overhead for splitting the iteration space and sharing or re-distributing work items. This cost is increased when scheduling overhead is comparable to the time needed to execute one iteration of the loop. Options which are considered efficient for loop parallelisation involve OpenMP's guided scheduling strategy and the more advanced work-stealing technique, implemented in frameworks like Intel®Cilk™Plus.

In this chapter we present a parallel for-loop scheduler which is based on work-stealing principles but runs under a completely cooperative scheme. POSIX signals are used by idle threads to interrupt left-behind workers, which in turn decide what portion of their workload can be given to the requester. We call this scheme Interrupt-Driven Work-Sharing (IDWS). This chapter describes how IDWS works, how it can be integrated into any POSIX-compliant OpenMP implementation and how a user can manually replace OpenMP parallel for-loops with IDWS in existing POSIX-compliant C++ applications. Additionally, we measure its performance using both a synthetic benchmark with varying distributions of workload across the iteration space and a real-life application on Intel®Xeon® Sandy Bridge and Intel®Xeon Phi™ systems. Regardless the workload distribution and the underlying hardware, IDWS is always the best or among the best-performing strategies, providing a good all-around solution to the scheduling-choice dilemma.

7.1 Introduction

Most parallelism in shared-memory parallel programming comes from loops of independent iterations, *i.e.* iterations which can be safely executed in parallel. However, distributing the iteration space over the available computational resources of a system is not always a simple thing. Fine-grained control of distribution is often associated with high overhead whereas static partitioning of the iteration space can lead to significant load imbalance. In both cases, the impact on performance is serious.

Research on an advanced for-loop scheduler was motivated by our work on PRAgMaTIc. Profiling data revealed that many of PRAgMaTIc's parallel loops are highly diverse, involving irregular computations which introduce high levels of iteration-to-iteration load imbalance. Existing scheduling strategies provided by OpenMP fail to achieve good balance with low scheduling overhead, whereas adaptive mesh algorithms which constantly modify mesh topology make it impossible to balance workload a priori.

We wanted the new scheduler to be portable and easily plug-able into the widely-adopted OpenMP API, so that it can target an as wide as possible range of systems, like Fujitsu's PRIMEHPC FX10, a SPARC64™-based supercomputer [43]. Those portability requirements prohibit the use of platform- or vendor-specific threading mechanisms, parallel libraries and language extensions, like Intel®Cilk™Plus [99, 42, 13]. On the contrary, they call for a POSIX-compliant implementation, based on the fact that most operating systems used in scientific computing are POSIX-compliant and most compilers (e.g. Linux versions of gcc, icc, xlc, etc.) implement OpenMP threads as POSIX threads (we have found it out by experimenting with those compilers). Of course, since every OS has threading and signalling mechanisms, the new scheduler can be implemented into any compiler on any OS.

The main contributions of this work are the following:

- Present an new interrupt-driven work-sharing scheduler (IDWS) which can easily be used with existing POSIX-compliant OpenMP applications.

- Demonstrate how OpenMP loops can be converted to IDWS loops.

- Describe how a compiler vendor can incorporate the new scheduler into their product.

- Show using a variety of benchmarks that IDWS is a good all around solution to the scheduling-choice dilemma, always being among the best-performing strategies in all benchmarks.

7.2 Background

OpenMP offers three different scheduling strategies for parallel loops: static, dynamic and guided [45]. There is also a more advanced scheduling technique, known as "work-stealing", which is implemented by libraries such as Intel®Cilk™Plus, though it is not part of the OpenMP specification, nor is it supported (to the best of our knowledge) by any OpenMP implementation. In this section we will present these four options and compare them in terms of load balance, scheduling overhead and overall efficiency.

7.2.1 OpenMP static

Under the static scheduling scheme, the iteration space is divided into equally large chunks which are then assigned to threads. This can be seen in the first example in Figure 7.1. Partitioning of iteration space is done statically at the beginning of the for-loop, so there is zero scheduling overhead. On the other hand, this scheme can lead to significant load imbalance, especially in a highly diverse loop.

7.2.2 OpenMP dynamic

Dynamic scheduling is a first approach to the problem of load imbalance. Instead of a static partitioning of the iteration space, chunks of work are assigned to threads dynamically. Once a thread finishes a chunk, it takes the next available from the iteration space. This is shown in the middle example in Figure 7.1. Access to chunks is done via atomic updates of the loop counter; a thread acquiring a chunk reads the current value of the loop counter and increments it atomically by the chunk size.

Dynamic scheduling solves imbalance problems as threads proceed to the next iterations of the for-loop in a fine-grained way. As an immediate consequence, good load balance comes at a cost. The loop counter is updated

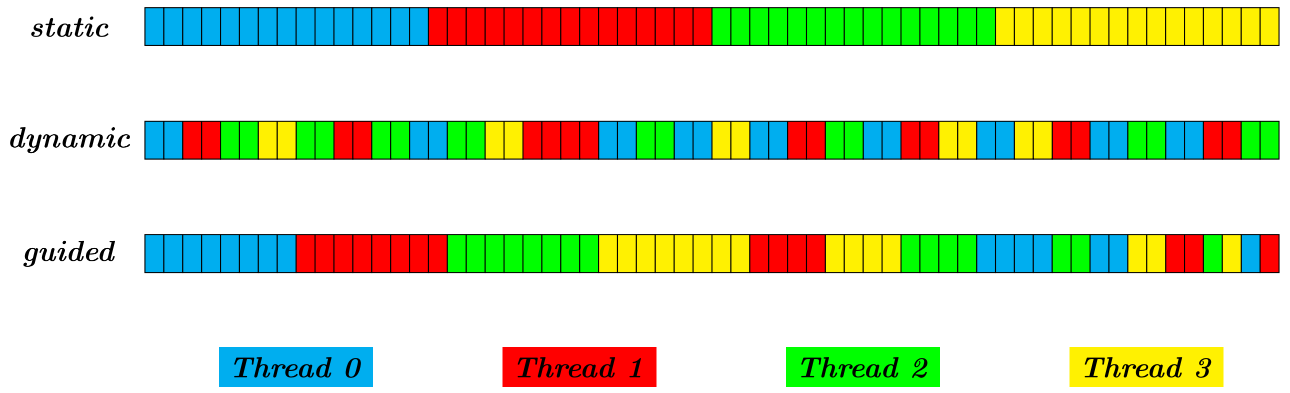


Figure 7.1: Example with four threads of the three scheduling strategies offered by OpenMP: static, dynamic (chunk=2) and guided. Note that under the guided scheme chuck size is reduced exponentially.

atomically and this constitutes a 2-way source of overhead. The two components of overhead are related to instruction latency and thread competition. The time it takes to execute an atomic instruction can vary anywhere between a standard update in L1 (if the thread performing the update is running on the same physical core as the thread which last updated the shared variable) and an update in RAM (if the last thread to update the shared variable is running on another socket in case of NUMA systems). This may not be a problem in short for-loops, but becomes easily a hotspot in loops with millions of iterations and little work per iteration (i.e. when atomic instruction latency is comparable to the loop body itself). Secondly, as the number of threads increases, so does the competition for the shared variable, leading to either (depending on the architecture) increased locking or increased number of failed update attempts, thus making atomic instruction latency even longer.

It could be argued that this overhead can be mitigated by increasing the chunk size, therefore lowering the number of times a thread will need to access the loop counter. On the other hand, increasing the chunk size can introduce load imbalance once again. Additionally, it is usually impossible to know the optimal chunk size at compile time and/or it can vary greatly between successive executions of an algorithm. Besides, relying on the chunk size for performance optimization puts an extra burden on the programmer.

We have found that using dynamic scheduling over guided in PRAgMaTIc can increase the execution time of specific algorithms by up to three times, as will be shown in Section 7.5. Following that, dynamic scheduling was rejected as an option for that framework.

7.2.3 OpenMP guided

The guided scheme is an attempt to reduce dynamic scheduling overhead while retaining good load balance. The key difference between the two strategies is how chunks of work are allocated. Whereas in dynamic scheduling the chunk size is constant, the guided scheme starts with large chunks and the size is reduced exponentially as threads proceed to subsequent iterations of the for-loop. This can be seen in the last example of Figure 7.1. Initial large chunks account for reduced atomic accesses to the loop counter while the more fine-grained control towards the end of the loop tries to maintain good load balance.

For the most part, guided scheduling works well in PRAgMaTIc, yet there are cases where we have observed significant load imbalance. This can happen if, for instance, most of the work in an irregular loop is accumulated in a few of the initial big chunks. In a case like that, threads processing the "loaded" chunks are busy for long while others go through the remaining "light" iterations relatively quickly and reach the end of the for-loop early, waiting for the busy workers to finish as well.

7.2.4 Work-stealing

Work-stealing ([12, 109]) is a more sophisticated technique aiming at balancing workload among threads while keeping scheduling overhead as low as possible. It has been shown that theoretically communication efficiency of work-stealing is "existentially optimal to within a constant factor" [14] compared with work-sharing techniques. The generic work-stealing algorithm for a set of tasks [12] can be summarized as follows. Each thread keeps a deque (double-ended queue) of tasks to execute. While the deque is full, the thread pops workitems from the front. Once the deque is empty, the thread becomes a thief, i.e. it chooses a victim thread randomly and steals a chunk of workitems from the back of the victim's deque.

For a parallel for-loop with a predefined number of iterations N the deques can simply be replaced with pairs of indices $< i_{start}, i_{end} >$ corresponding to the range in the iteration space $[i_{start}, i_{end})$ which has been assigned to each thread, $0 \leq i_{start}, i_{end} < N$. In this case, every thread executes iterations by using i_{start} as the loop counter whereas thieves steal work by decrementing a victim's i_{end}.

Accesses to those pairs of indices can lead to race conditions, so they need to be accessed with atomics. Following that, work-stealing for for-loops comes close to OpenMP's dynamic scheduling with some chunk size > 1, with a major difference being that in work-stealing threads do not compete all together for atomic access to the same shared variable (the common loop counter); instead, congestion is rare and happens only if two thieves try to steal from the same victim.

Performance can still suffer from load imbalance and scheduling overhead when using work-stealing. The main drawback of the classic work-stealing algorithm is that thieves choose victims randomly. There is no heuristic method to indicate which threads are more suitable victims (i.e. have more remaining workload) than others. Stealing comes at a cost and picking victims with too little or no remaining work is inefficient, as it leads to the need for frequent stealing which induces some overhead. Additionally, failed attempts do not help balance the workload. As an example of an extreme case, a single thread becomes the sole remaining worker while the rest waste time trying to steal from each other in vain.

Mitigating the effects of random choice was our main concern when designing the new for-loop scheduler. We devised a low-overhead heuristic method for finding appropriate victims. At the same time, we tried to reduce scheduling overhead by eliminating the need to use atomics when accessing each thread's $< i_{start}, i_{end} >$ pair. The following section describes in detail how the scheduler is implemented.

7.3 Interrupt-Driven Work Sharing

Our new scheduler differs from existing work-stealing approaches in two major ways. First of all, as was mentioned in Section 7.2.4, every worker constantly "advertises" its progress so that thieves can find suitable victims which have been left behind. Secondly, a thief does not actually steal work from the victim in the classic sense; instead, it interrupts the chosen victim by sending a POSIX signal. The signal handler executed by the victim encapsulates the code with which the victim decides what portion of its remaining workload can be given away. This interrupt technique is an instance of an asymmetric Dekker protocol [32]. Using asynchronous direct messages for fine-grained parallelism was proposed by Sanchez *et al.* [103], albeit at the hardware level; our implementation, on the other hand, is solely based on existing software tools and support by the operating system.

As it becomes apparent, the new scheduling algorithm is much closer to work-sharing than work-stealing, therefore we call it Interrupt-Driven Work-Sharing (IDWS). Nonetheless, we will use work-stealing terminology throughout this chapter in order to be consistent with the literature and avoid creating confusion.

The abstract description of this scheme can be split into three parts:

Algorithm 20 Parallel loop executed by each thread

```
for $i = i_{start}$; $i < i_{end}$; $i \leftarrow i + 1$ do
    flush $i$                    ▷ from register file to memory, so that
                                 ▷ thieves can see this thread's progress
    execute $i_{th}$ iteration

    flush $i_{end}$              ▷ from memory to register file, as it may
                                 ▷ have been modified by the signal handler
end for
```

Algorithm 21 Work-stealing

```
for all threads $t_n$ do
    $remaining_{t_n} \leftarrow i_{end,t_n} - i_{start,t_n} - i_{t_n}$
end for
let T $\leftarrow t_n$ for which $remaining_{t_n} = max$
send signal to victim T
wait for answer
update own $i_{start}, i_{end}$
execute loop chunk
```

Algorithm 22 Signal handling

```
$remaining \leftarrow i_{end} - i_{start} - i$
if $remaining > 1$ then
    $chunk \leftarrow remaining/2$
    $i_{end,thief} \leftarrow i_{end}$
    $i_{start,thief} \leftarrow i_{end} - chunk$
    $i_{end} \leftarrow i_{end} - chunk$
end if
send reply to thief
```

Loop execution (Algorithm 20) Every thread executes the iterations of the chunk it has acquired in the same way as it would using OpenMP's static scheduling scheme. Initially, the iteration space is divided statically into chunks of equal size and every thread t_n is assigned one chunk. The chunk's boundaries for thread t_n, referred to as i_{start,t_n} and i_{end,t_n}, are globally visible variables accessible by all threads. Compared to static scheduling, the important addition here is some necessary flushing of the loop counter i_{t_n} and the loop boundary i_{end,t_n}. More precisely, the value of i_{t_n} has to be written back to memory (instead of being cached in some register) at the beginning of every iteration so that potential thieves can monitor

t_n's progress, calculate how much work is left for t_n and decide whether it is worth stealing from it. Similarly, the end boundary $i_{end,tn}$ has to read by t_n from memory (instead of caching it in some register) before proceeding to the next iteration because i_{end,t_n} might have been modified by the signal handler if a thief interrupted t_n while the latter was executing an iteration of the for-loop.

Choosing suitable victims (Algorithm 21) By flushing their loop counters, threads advertise their progress so potential thieves can find where to steal from. When a thread becomes a thief, it calculates the remaining workload for all other threads by reading the associated values i_{t_n}, i_{start,t_n} and i_{end,t_n}. This way, we have a heuristic method for finding which thread has the most remaining work, thus being a more suitable victim than others. Stealing from the most loaded thread is quite an old idea, dating back to the work by Markatos & LeBlanc [80] and Subramaniam & Eager [107] on affinity scheduling. This heuristic may not be optimal, but is an improvement over random choice. Once the thief has spotted its victim, it sends a signal and waits for an answer. The victim executes the signal handler and replies with the boundaries (a pair of $< i_{start}, i_{end} >$) of the chunk it wants to give away. Finally, the thief becomes a worker once again and moves on to process the newly acquired chunk.

Signal handler (Algorithm 22) When a victim is interrupted by the signal, control is transferred by the operating system to the associated handler. Inside that code, the thread calculates how much work it can give away (if any), replies to the thief with the boundaries of the donated chunk, re-adjusts the boundaries of its own chunk and finally returns from the signal handler. It has been shown that the *steal-half* strategy [54], *i.e.* stealing half the remaining workitems from the victim, is more effective than stealing different percentages [25].

It is clear that there are no races and no need for atomic access to any loop variables during the stealing process, as the donor is the one who decides what will be donated. Of course, switching from user to kernel mode to execute the signalling system call and busy-waiting for a reply from the victim involves some overhead; however, as will be shown in the results section, this method seems to be more efficient than classic work stealing.

7.4 C++ implementation and usage

This section describes how the IDWS scheduler is implemented and how it can replace existing OpenMP for-loops.

7.4.1 IDWS namespace

IDWS is a namespace encapsulating all necessary data structures and functions used by the new scheduler. Its declaration can be seen in Code Snippet 4.

```
namespace IDWS{
  struct thread_state_t;
  vector<thread_state_t> thread_state;
  void SIGhandlerUSR1(int sig);
  void IDWS_Initialize();
  void IDWS_Finalize();
};
```

Code Snippet 4: IDWS namespace. It consists of initialisation and finalisation functions, the signal handler, the definition of `struct thread_state_t` and the vector holding all `thread_state_t` instances (one per thread).

```
struct thread_state_t{
  size_t start;
  size_t end;
  size_t processed;
  int current_ctx;
  bool active;

  int signal_arg;
  pthread_t ptid;
  pthread_mutex_t comm_lock;
  pthread_mutex_t request_mutex;
  pthread_cond_t wait_for_answer;
};
```

Code Snippet 5: thread_state_t struct

struct `IDWS::thread_state_t` The heart of IDWS is a data structure named `thread_state_t`, which encapsulates all variables involved in parallel loop execution and work-stealing. Each participating thread has its own instance of this struct, which is accessible by all other threads. The struct can be seen in Code Snippet 5.

- `start` and `end`: Define the current chunk boundaries.
- `processed`: Is used by a thread to advertise its progress through the loop.
- `current_ctx`: IDWS loops are `nowait` loops, which means that a thread can proceed to the rest of the program without synchronising with other threads. In order to know whether two threads work inside the same loop, so stealing work from one another is valid, a counter `current_ctx` is used, which is incremented each time a thread finishes a loop. Here we assume that all threads will go through all loops of the program.
- `active`: Indicates whether the thread is inside the loop; this variable is used by thieves to skip immediately threads which have also become thieves.
- `signal_arg`: POSIX signals can only have two arguments, what signal is to sent and to whom. The victim needs to know, however, who the thief is, so `signal_arg` is used by the thief to send its ID to the victim.
- `comm_lock`: In order to avoid needless busy-waiting by other thieves while one thief has already sent a signal to its victim, we use a lock (in form of a mutex); while this lock is held by a thief, other thieves will choose other victims to steal from.
- `ptid`: POSIX ID of the thread; it is used by the thief to raise the signal.
- `request_mutex` and `wait_for_answer`: POSIX mutex and condition variables which assist the process of sending the signal and waiting for a reply. Locking the mutex also serves as a memory fence so that the victim is guaranteed to see the arguments sent by the thief.

Note that we need two separate mutexes and cannot use `request_mutex` in place of `comm_lock`. The former is implicitly released by the thief in order to enable the victim to signal the condition variable; in the meantime, before the victim locks `request_mutex`, another thief might acquire the lock and destroy the process.

vector `IDWS::thread_state` Each thread has its own instance of the `thread_state_t` struct. All instances are held in a shared vector called `thread_state`.

Initialisation and finalisation Like MPI, IDWS needs to be initialised by calling `IDWS::IDWS_Initialize()`. During initialisation, threads create their `thread_state_t` structs and push them back into the shared vector `thread_state`. Struct initialisation also includes finding POSIX IDs and initialising `comm_lock`, `request_mutex` and `wait_for_answer`. Similarly, this data has to be destroyed at the end of the program, which is done by a call to `IDWS::IDWS_Finalize()`. Additionally, we must register a signal handler to serve the interrupt. We have chosen signal `SIGUSR1` and function `IDWS::SIGhandlerUSR1` as the signal handler. Choice of `SIGUSR1` was arbitrary; it should be noted, however, that if an application uses the same signal for other purposes, it must re-register the original handler upon finishing with IDWS or use a different signal in the first place.

Signal handler A victim decided what portion of its chunk can be donated by executing the signal handler. The way it is done is described in Code Snippet 6. The victim first checks that the thief works in the same context. Then, it calculates how much work it can give away using `start`, `end` and `processed`, also leaving a safety margin due to an uncertainty regarding the true value of `processed`. In case of success, the victim updates both the thief's and its own `start` and `end` and sets `sig_arg=1` to indicate successful donation (otherwise, `sig_arg` is set to another value). Finally, the victim signals the condition variable to let the thief know that the signal handler is over.

7.4.2 Prologue and epilogue macros

The new scheduler is defined in two parts, using macros `IDWS_prologue` and `IDWS_epilogue`. These macros must surround the loop body.

```c
void SIGhandlerUSR1(int sig){
  int tid = omp_get_thread_num();
  // Who sent the signal
  int sig_thread = thread_state[tid].signal_arg;
  pthread_mutex_lock(&thread_state[tid].request_mutex);

  // Only share a chunk if both
  // threads are in the same context
  if(thread_state[tid].current_ctx ==
    thread_state[sig_thread].current_ctx){
    size_t remaining = thread_state[tid].end -
      thread_state[tid].start - thread_state[tid].processed;
    // Leave a safety margin - we do not know if the
    // signal was caught before, after or even in the
    // middle of updating thread_state[tid].processed.
    if(remaining > 0){
      --remaining;
      size_t chunk = remaining/2;
      thread_state[sig_thread].start =
        thread_state[tid].end - chunk;
      thread_state[sig_thread].end = thread_state[tid].end;
      thread_state[tid].end -= chunk;
      // reply success
      thread_state[tid].signal_arg = -1;
    }else
      thread_state[tid].signal_arg = -2; // reply failure
  }else
    thread_state[tid].signal_arg = -2; // reply failure

  pthread_cond_signal(&thread_state[tid].wait_for_answer);
  pthread_mutex_unlock(&thread_state[tid].request_mutex);
}
```

Code Snippet 6: Signal handler.

`IDWS_prologue` macro Before entering a loop, the iteration space is split into equal chunks which are assigned to threads. After that, each thread begins the execution of its chunk. Compared to a standard for-loop, a IDWS for-loop is defined slightly differently. Apart from checking for the end of the loop and incrementing the counter after every iteration, in IDWS we must also enforce the compiler to flush the counter back to memory and load the updated value of i_{end} from memory (which might have been modified by the signal handler), as indicated by Algorithm 20. Flushing is done selectively for those two variables by casting them to volatile datatypes. Using `#pragma omp flush` would flush the entire shared program state, which is not efficient. A pseudo-code of how the macro expands is given in Code Snippet 7. Parameters `TYPE`, `NAME` and `SIZE` correspond to the datatype of the loop counter, its name and the size of the iteration space,

```
/* IDWS_prologue(TYPE,NAME,SIZE) starts expanding here */
  // assume tid = omp_get_thread_num();
  thread_state[tid].start = ...;
  thread_state[tid].end = ...;
  thread_state[tid].processed = 0;
  thread_state[tid].active = true;

  do{
    size_t __IDWS_cnt= 0;
    for(TYPE NAME = thread_state[tid].start; ; ++NAME, ++__IDWS_cnt){
      // Force flushing the progress back into memory
      *((volatile size_t*) &thread_state[tid].processed) = __IDWS_cnt;
      // Force re-loading the end boundary from memory
      if(NAME >= *((volatile size_t*) &thread_state[tid].end))
        break;
/* IDWS_prologue ends here */

      /****************************
       * loop body is executed here *
       ****************************/

/* IDWS_epilogue starts expanding here */
    } // end for

    // become a thief
    thread_state[tid].active = false;
    std::map<int,size_t> remaining;
    forall(t in active threads) // only check non-thieves
      remaining[t] = thread_state[t].end - thread_state[t].start -
        thread_state[t].processed;
    traverse remaining from largest to smallest;
    victim = first thread t for which
      pthread_mutex_trylock(&thread_state[t].comm_lock) succeeds;
    if(no victim found)
      break; // exit the do-while loop

    // tell the victim who we are
    thread_state[victim].sig_arg = tid;
    pthread_mutex_lock(&thread_state[victim].request_mutex);
    pthread_kill(thread_state[victim].ptid, SIGUSR1); // send signal
    pthread_cond_wait(&thread_state[victim].wait_for_answer,
      &thread_state[victim].request_mutex);
    pthread_mutex_unlock(&thread_state[victim].request_mutex);

    // become a worker again
    if(thread_state[victim] == -1) thread_state[tid].active = true;
    pthread_mutex_unlock(&thread_state[victim].comm_lock);
  } while(thread_state[tid].active = true) // end do

  thread_state[tid].current_context++; // proceed to next loop
/* IDWS_epilogue ends here */
```

Code Snippet 7: Pseudo-code demonstrating how `IDWS_prologue` and `IDWS_epilogue` are expanded around the loop body.

respectively. In the current implementation of the new scheduler we assume that loops run from 0 to `SIZE` with increments of 1 and that the loop counter is of an unsigned integral datatype.

`IDWS_epilogue` macro After a thread finished its chunk, it becomes a thief. That means it has to enter the stealing process, as described in Algorithm 21. The `IDWS_epilogue` macro serves this purpose. The way the macro expands can be seen in Code Snippet 7. The thief calculated for all active workers the amount of remaining work. Then, starting from the worker with the highest remaining workload, it tries to acquire the worker's `comm_lock`. If no suitable worker is found, then the thief exits the IDWS loop and proceeds to the rest of the code. Otherwise, the thief locks the victim's mutex, sends the signal and waits on the victim's condition variable for an answer. The answer comes via `sig_arg`. If `sig_arg==-1`, then the victim has set the thief's `start` and `end` variables, so the thief becomes a worker again. If any other answer has been sent back, then the thief exits the IDWS loop. It is important to note that a memory fence is necessary on the thief's side between setting the victim's signal argument `sig_arg` and raising the signal, so that we make sure that the victim will see the correct value of `sig_arg`. Locking the victim's mutex before sending the signal works as an implicit memory fence.

7.4.3 OpenMP to IDWS

```c
#include <omp.h>
...
int main(){
  ...
  #pragma omp parallel
  {
    ...
    #pragma omp for
    for(TYPE VAR=0; VAR<SIZE; ++VAR){
      do_something(VAR);
    }
    ...
  }
  ...
}
```

Code Snippet 8: Initial OpenMP for-loop. The loop must be inside an OMP parallel region.

```
#include <omp.h>
#include "IDWS.h"
...
int main(){
  ...
  IDWS::IDWS_Initialize();
  int nthreads = omp_get_max_threads();
  ...
  #pragma omp parallel
  {
    int tid = omp_get_thread_num();
    ...
    IDWS_prologue(TYPE, VAR, SIZE)
      do_something(VAR);
    IDWS_epilogue
    ...
  }
  ...
  IDWS::IDWS_Finalize();
}
```

Code Snippet 9: Transformed code showing what has to be added/modified in order to use the new scheduler instead of a standard OpenMP scheduling strategy.

The new scheduler can be used directly with virtually any C++ OpenMP application written for any POSIX-compliant operating system (provided that the compiler used implements OpenMP upon pthreads). A prerequisite for converting an OpenMP loop to a IDWS one is that the former is written as shown in Code Snippet 8, i.e. the loop must be inside an `omp parallel` region. Conversion to IDWS loops is shown in Code Snippet 9. The user needs to include header file "IDWS.h" which can be downloaded from PRAgMaTIc's page on Launchpad[1]. This header file defines the IDWS namespace and the prologue and epilogue macros.

Compared to the initial version, we need to define:

- `int nthreads=omp_get_max_threads()`: shared variable outside the parallel region,
- `int tid=omp_get_thread_num()`: thread-private variable inside the parallel region,

remove the `#pragma omp for` directive and the for-loop declaration and, finally, surround the loop-body with the `IDWS_prologue` and `IDWS_epilogue`

[1]`https://code.launchpad.net/~gr409/pragmatic/IDWS`

macros.

7.5 Experimental Results

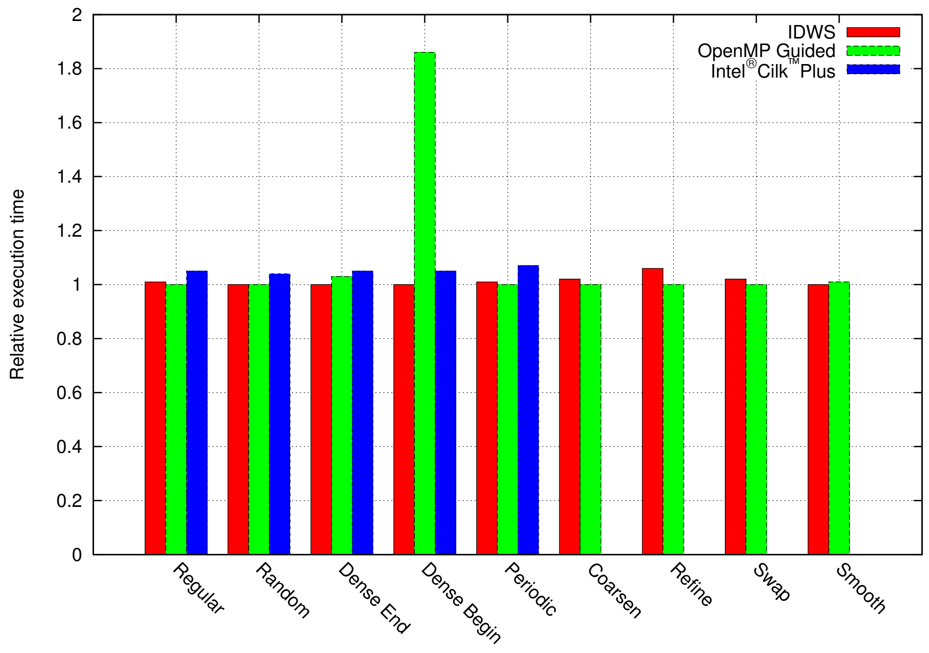


Figure 7.2: Relative execution time between IDWS, OpenMP guided and Intel®Cilk™Plus on the Intel®Xeon® E5-2650 system. For each benchmark, the fastest scheduling strategy is taken as reference (scoring 1.0 on the y-axis).

In order to measure the performance of our new scheduler, we ran a series of tests using both synthetic benchmarks and real kernels from the PRAgMaTIc framework. We used three systems: a dual-socket Intel®Xeon® E5-2650 (Sandy Bridge, 2.00GHz, 8 physical cores per socket, 2 hyperthreads per core, 32 threads in total), a dual-socket Intel®Xeon® E5-2643 (Sandy Bridge, 3.30GHz, 4 physical cores per socket, 2 hyperthreads per core, 16 threads in total) and an Intel®Xeon Phi™ 7120P board (1.238GHz, 61 physical cores, 4 hyperthreads per core, 244 threads in total). Both Intel®Xeon® systems run Red Hat®Enterprise Linux® Server release 6.4 (Santiago). Both versions of the code (intel64 and mic) were compiled with Intel®Composer XE 2013 SP1 using the `-O3 -xAVX` optimisation flags. The

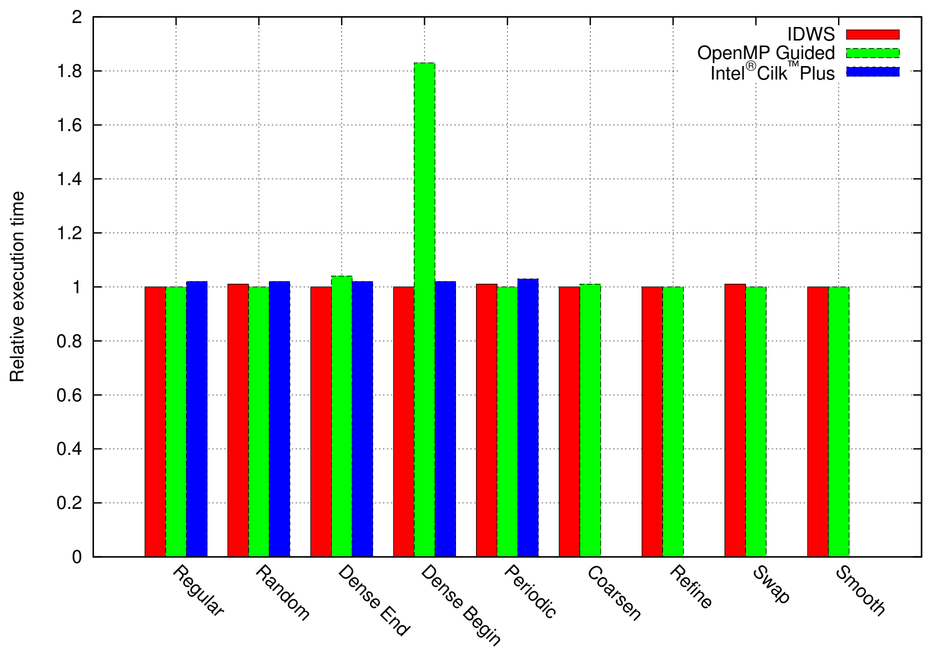


Figure 7.3: Relative execution time between IDWS, OpenMP guided and Intel®Cilk™Plus on the Intel®Xeon® E5-2643 system. For each benchmark, the fastest scheduling strategy is taken as reference (scoring 1.0 on the y-axis).

benchmarks were run using Intel®'s thread-core affinity support with the maximum number of available threads on each platform. Additionally, we ran a second series of benchmarks on Intel®Xeon Phi™ using half the available number of threads (61 cores, 2 hyperthreads per core) in order to link this section to PRAgMaTIc's performance results discussed in Section 6.5 and more specifically in order to highlight the contribution of IDWS to the problem we observed when running PRAgMaTIc with guided scheduling using 244 threads.

The synthetic benchmark was designed to be compute-bound with minimal memory traffic and no thread synchronization. Our purpose is to show how the different scheduling strategies compare to each other in terms of achievable load balance and incurred scheduling overhead without being affected by other factors (such as memory bandwidth, data locality etc.). The synthetic benchmark uses an array `int states[16M]`, which is populated with values in the range [0..3]. Then, the parallel loop iterates over this ar-

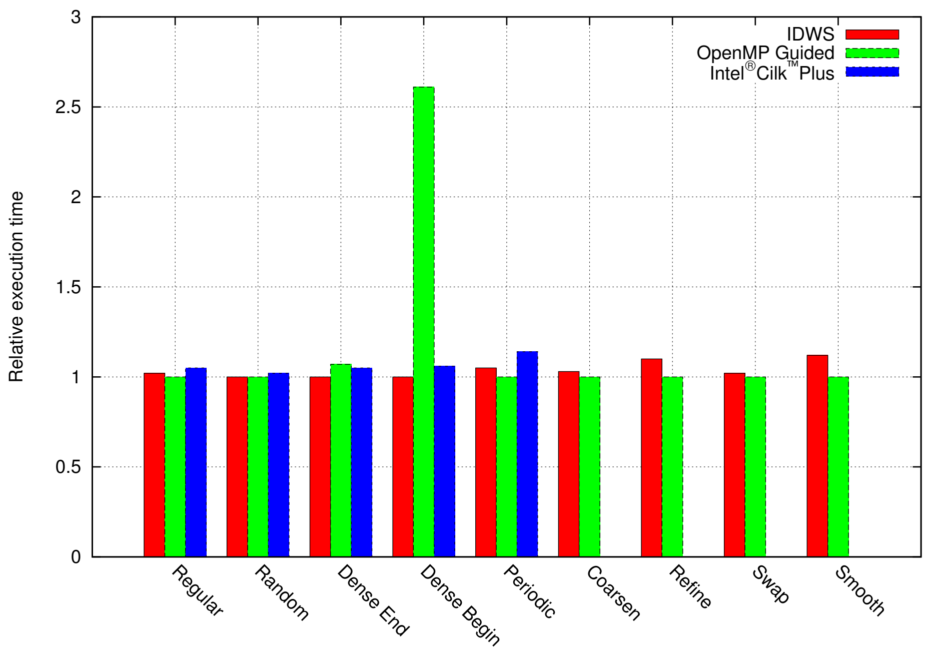


Figure 7.4: Relative execution time between IDWS, OpenMP guided and Intel®Cilk™Plus on the Intel®Xeon Phi™ 7120P coprocessor using 122 threads. For each benchmark, the fastest scheduling strategy is taken as reference (scoring 1.0 on the y-axis).

ray. For each element i, $i \in [0..16M)$, the kernel performs a different amount of work according to the value of `states[i]`. If `states[i]==0`, nothing is done. If `states[i]==1`, the kernel computes sin() values of `i` and powers of `i`. If `states[i]==2`, the kernel additionally computes cos() values of `i` and its powers. Finally, if `states[i]==3`, the kernel additionally computes some sinh() values.

Array `states` is populated five times with different distributions of workload and total amount of work. Each population has been given a name:

- Regular: All elements of `states` are set equal to 2. This is a distribution corresponding to a regular loop which does the exact same thing in every iteration. Load imbalance is still possible, as interference from the OS and other factors (e.g. cache conflicts, hyper-threading issues) can hold some threads behind.

- Random: `states` is populated with random values following a uni-

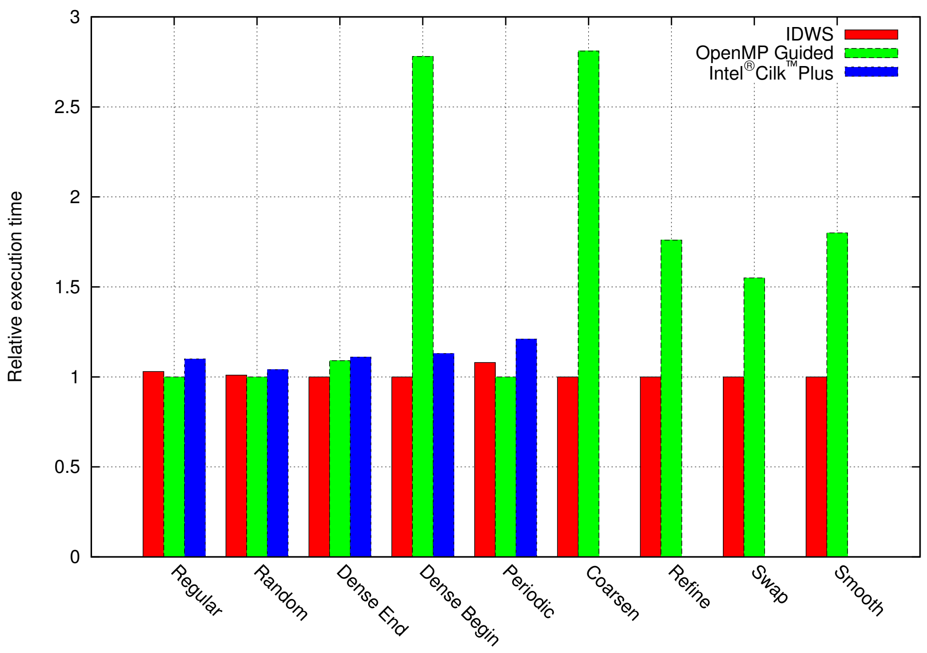


Figure 7.5: Relative execution time between IDWS, OpenMP guided and Intel®Cilk™Plus on Intel®Xeon Phi™ 7120P coprocessor using 244 threads. For each benchmark, the fastest scheduling strategy is taken as reference (scoring 1.0 on the y-axis).

form distribution. This sub-benchmark corresponds to real-life distributions in problems like graph colouring or the swap and smooth kernels in PRAgMaTIc.

- Dense End: Most of the workload is accumulated towards the end of the iteration space, where `states[i]=3`, while the beginning is populated with a uniform mixture of values [0..3). The rest of the iteration space is set to 0, i.e. no work. This is a distribution closely related to the refinement kernel in PRAgMaTIc.

- Dense Start: Mirrored distribution of Dense End. Closely related to PRAgMaTIc's coarsening kernel. This is an example of workload distribution for which OpenMP guided scheduling is a bad choice.

- Periodic: There is a repeating pattern of states throughout the iteration space. It is particularly bad for static scheduling with interleaved allocations of iterations (i.e. with some chunk size).

Apart from the synthetic benchmark, we also ran PRAgMaTIc using the various scheduling options in order to see how each strategy performs in a real-life scenario, where compute capacity is not the only performance-limiting factor. It should be noted that PRAgMaTIc is build upon OpenMP, so there are no results for Intel®Cilk™Plus in this case.

Table 7.1, Table 7.2 and Tables 7.3 & 7.4 show the execution time on the three platforms, respectively, using six scheduling strategies for each distribution of the synthetic benchmark and the four PRAgMaTIc kernels. The strategy named "OMP static,1" is static scheduling with chunk size equal to 1. As can be seen, IDWS is either the fastest scheduling option or very close to the fastest for each benchmark-platform combination. Additionally, it clearly outperforms Intel®Cilk™Plus, with the performance gap becoming wider as the number of threads increases and Cilk's design to pick victims in a random fashion becomes inefficient. Those results make IDWS look promising for the thousand-core era.

Regarding PRAgMaTIc, the major competitor of IDWS seems to be OpenMP's guided scheduling. Despite not being very suitable for certain kernels (coarsening) theoretically, in practice it performs just as well as IDWS. A notable exception is the 244-thread case on Intel®Xeon Phi™, where guided scheduling is the worst choice among the available options. Continuing the discussion from Section 6.5 regarding this case, we see that IDWS gives a considerable boost to performance, resulting in execution times much closer to the expected ones. Results for the 244-thread case are still worse that those for 122 threads, but this is due to the reasons we analysed in the related paragraph on PRAgMaTIc's performance.

A comparison of relative performance between the three major competitors (IDWS, OpenMP guided and Intel®Cilk™Plus) is shown in Figure 7.2 (Intel®Xeon® E5-2650 system), Figure 7.3 (Intel®Xeon® E5-2643 system), Figure 7.4 (Intel®Xeon Phi™ with 122 threads) and Figure 7.5 (Intel®Xeon Phi™ with 244 threads). For each benchmark, we compare the relative execution time between IDWS, OpenMP guided and Intel®Cilk™Plus (for PRAgMaTIc kernels there is only IDWS vs OpenMP guided comparison). Reference execution time per benchmark, i.e. the one which corresponds to 1.0 on the y-axis, is execution time of the fastest scheduler.

Finally, in order to back up our claim that random choice of victims can hinder the efficiency of work-stealing, we ran the synthetic benchmarks using a modified version of IDWS in which thieves do not use a heuristic to pick their victims; instead, the decision is random. A comparison of relative performance of the two IDWS versions on Intel®Xeon® E5-2650 as a function of the number of threads is shown in Figure 7.6. It is clear that not having a heuristic has a serious impact on the efficiency of work-stealing as the number of threads increases. Picking the wrong victim incurs the overhead of mutex- and signal-related system calls on the thief's side and the interruption of the victim. Had we used a work-stealer based on atomics, the overhead of picking the wrong victim would be lower (atomics are definitely faster than system calls); yet it would still be there, as the thief would waste time trying to hit the right victim instead of getting a chunk of work as soon as possible.

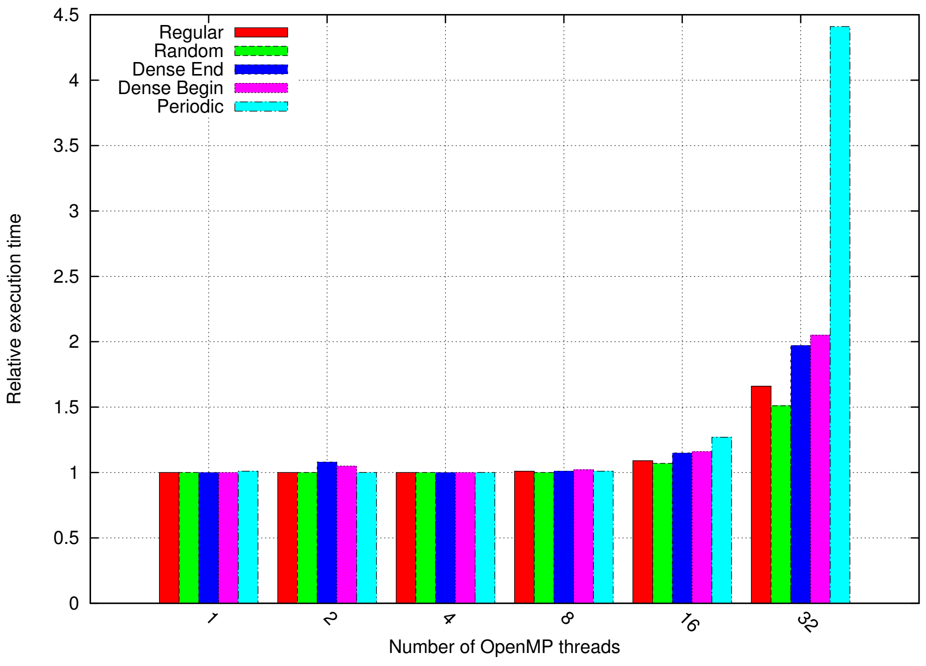


Figure 7.6: Relative execution time between regular IDWS and the version in which victims are chosen randomly on Intel®Xeon® E5-2650. Execution time of regular IDWS is taken as reference (1.0).

7.6 Study of Potential Overheads

There are aspects of our scheduler which can be claimed to introduce overheads not present in classic work-stealing implementations. Below, we list potential sources of overhead and argue that they do not impact performance in critical ways.

- *Calculating the remaining workload for all potential victims is a linear function of the number of participating threads and this approach might not be scalable to thousands of cores.* However, our profiling results showed that workload calculation is not a performance bottleneck, even on Intel®Xeon Phi™ with 244 threads, and the actual hotspot is waiting for an answer from the victim.

- *The thief interrupts a victim, which executes a handler to give away work. This interruption, and the corresponding handling routine, affects the critical path of a worker, adding overhead to its execution. It is the idle thread the one that should perform the corresponding chores.* Admittedly, the victim's critical path is affected when it executes the signal handler but in exchange the victim can access its loop variables very fast, without atomics. It is a design compromise which seems to work.

- *IDWS workers expose their progress by constantly flushing to/from memory their loop counter and their upper bound. This is a source of inefficiency.* Constant flushing is not an overhead on modern hardware. Valid copies of the loop counter and upper bound are always in the victim's L1 cache, since only the victim modifies them, so load-/store latency is only 4-6 cycles. On modern systems with pipelined architectures, instruction re-ordering by the compiler, out-of-order execution and speculative execution of branches, the total overhead is at most 2-3 cycles. That is really negligible compared to the loop body, which can be from tens up to millions of cycles long. In CPUs with superscalar or heterogeneous pipelines (*i.e.* virtually everything on the market since the mid 1990s), which issue more than one instructions in parallel, the overhead can easily be zero cycles - loads/stores just fill pipeline slots which would remain empty otherwise.

7.7 Related Work

A communication-based work-stealer was developed by Acar *et al.* [2]. It differs from our scheduler in numerous ways. Most notably, this work-stealer uses atomics instead of signals to notify the victim of the steal request: each thread has a mailbox and potential thieves write their IDs atomically using compare-and-swap operations when they want to steal work. The victim checks its mailbox at the end of every iteration of the for-loop and reads the ID of the thief who (if any) wrote to the mailbox. This can be wasteful for the thief, who has to wait until the victim completes the current iteration of its loop; in our implementation the victim is interrupted immediately, minimising the thief's waiting time. Additionally, in Acar's version victims are picked randomly and there is no heuristic for finding the most loaded worker.

7.8 Conclusions

This chapter described the Interrupt-Driven Work-Sharing for-loop scheduler which is based on work-stealing principles and tries to address a major problem of the original work-stealing algorithm: random choice of victims. The first implementation of IDWS was shown to work very efficiently, outperforming Intel®Cilk™Plus, while being from slightly slower to considerably faster than the best (per kernel) OpenMP scheduling strategy. These results indicate that IDWS could become the universal default scheduler for OpenMP for-loops, freeing the programmer from tricky and disruptive management of load balance.

	Synthetic benchmark					PRAgMaTIc kernels			
	Regular	Random	Dense End	Dense Begin	Periodic	Coarsen	Refine	Swap	Smooth
IDWS	5.85	7.48	4.14	3.80	1.13	12.4	7.29	19.9	11.0
OMP static	5.84	7.51	15.7	15.0	1.13	18.6	7.87	20.4	12.1
OMP static,1	5.92	7.60	4.32	3.91	14.5	15.4	8.45	22.5	12.4
OMP dynamic	22.4	27.0	21.4	20.5	9.72	67.4	31.4	99.7	17.9
OMP guided	5.82	7.46	4.27	7.08	1.12	12.1	6.88	19.5	11.1
Cilk	6.12	7.76	4.35	4.00	1.20	-	-	-	-

Table 7.1: Execution time in seconds for each benchmark using the 6 different scheduling strategies on a dual-socket Intel®Xeon® E5-2650 (Sandy Bridge, 2.00GHz, 8 physical cores per socket, 16 hyperthreads per socket, 32 threads in total).

	Synthetic benchmark					PRAgMaTIc kernels			
	Regular	Random	Dense End	Dense Begin	Periodic	Coarsen	Refine	Swap	Smooth
IDWS	8.24	10.6	5.86	5.37	2.86	17.5	7.06	24.5	17.3
OMP static	8.25	10.6	21.2	22.7	2.86	29.3	8.13	26.8	18.3
OMP static,1	8.37	10.7	6.01	5.70	22.2	23.0	9.55	30.7	19.1
OMP dynamic	26.9	34.6	22.8	21.7	9.43	63.7	26.3	91.8	22.7
OMP guided	8.23	10.5	6.07	9.81	2.84	17.6	7.08	24.3	17.3
Cilk	8.38	10.7	5.96	5.47	2.92	-	-	-	-

Table 7.2: Execution time in seconds for each benchmark using the 6 different scheduling strategies on a dual-socket Intel®Xeon® E5-2643 (Sandy Bridge, 3.30GHz, 4 physical cores per socket, 8 hyperthreads per socket, 16 threads in total).

	Synthetic benchmark					PRAgMaTIc kernels			
	Regular	Random	Dense End	Dense Begin	Periodic	Coarsen	Refine	Swap	Smooth
IDWS	11.0	19.6	9.60	9.02	0.97	30.7	17.2	86.7	26.9
OMP static	12.3	22.3	49.0	54.2	1.06	34.7	19.7	79.1	27.7
OMP static,1	12.6	22.9	11.2	10.6	48.3	35.6	21.2	122	26.2
OMP dynamic	40.1	31.5	25.7	25.1	15.2	129	59.6	234	29.4
OMP guided	10.8	19.6	10.3	23.5	0.92	29.9	15.6	85.3	24.0
Cilk	11.3	19.9	10.1	9.59	1.05	-	-	-	-

Table 7.3: Execution time in seconds for each benchmark using the 6 different scheduling strategies on Intel®Xeon Phi™ (1.2GHz, 61 physical cores, 2 hyperthreads per core, 122 threads in total).

	Synthetic benchmark					PRAgMaTIc kernels			
	Regular	Random	Dense End	Dense Begin	Periodic	Coarsen	Refine	Swap	Smooth
IDWS	7.46	15.9	7.46	7.06	0.56	34.2	19.9	177	25.9
OMP static	8.13	16.5	27.0	27.1	0.51	29.1	21.3	174	19.4
OMP static,1	7.65	16.0	7.63	7.24	24.7	27.9	20.0	202	19.3
OMP dynamic	17.3	19.8	13.9	13.6	7.68	81.4	38.4	247	24.4
OMP guided	7.27	15.7	8.11	19.6	0.52	96.1	35.1	275	46.6
Cilk	8.03	16.3	8.31	7.97	0.63	-	-	-	-

Table 7.4: Execution time in seconds for each benchmark using the 6 different scheduling strategies on Intel®Xeon Phi™ (1.2GHz, 61 physical cores, 4 hyperthreads per core, 244 threads in total).

8 Conclusions

In this chapter we review the achievements of this thesis which support the claims that are made within, putting an emphasis on the novel and important aspects of the investigations we have presented. At the end we present our ideas for future work and how it should proceed.

8.1 Summary of Thesis Achievements

In this section we present the achievements of this research, structured around the contributions list from Section 1.4.

- *We present an irregular compute framework consisting of scalable parallel techniques for manipulating mutable irregular data.* In Chapter 4 we analysed the topological hazards and data races that can occur when working on irregular data and argued that graph colouring combined with the deferred-updates strategy results in safe parallel execution. Moreover, we proposed the use of atomics to create shared worklists, which provides a synchronisation-free method compared to classic reduction-based worklist creation.

- *We demonstrate an improved parallel greedy colouring algorithm for shared-memory environments.* In Chapter 5 we reviewed older approaches to parallelising the greedy graph colouring algorithm and listed their weaknesses. Building upon the best among them, the algorithm by Çatalyürek *et al.*, we reduced performance-limiting barriers and devised an improved optimistic version which outperforms its predecessor by as much as 50% in heavily multithreaded environments. This case provides evidence that introducing thread divergence in an optimistic algorithm can reduce the amount of rolling-back.

- *We show how the scalable parallel techniques are applied to adaptive mesh algorithms.* Graph colouring, atomic-based worklist creation and the deferred-updates strategy were combined with the adaptive algorithms from Chapter 2, resulting in PRAgMaTIc, the first (to the best of our knowledge) threaded implementation of mesh adaptivity. Performance results in Chapter 6 demonstrate good scalability and parallel efficiency, given the invasive nature of adaptive algorithms, even in NUMA configurations.

- *We present some early work on an interrupt-based work-sharing scheduler for OpenMP.* In Chapter 7 we explained why built-in scheduling strategies of OpenMP are not optimal and proposed an interrupt-driven work-sharing scheduler which seems to be a good all-around option for OpenMP, while also outperforming Intel®Cilk™Plus.

8.2 Discussion

Contributions of this thesis can be split into two groups, (a) the algorithmic innovations and (b) their embodiment in software which led to the first effective threaded mesh adaptivity framework. Below we discuss weaknesses of our work and in the next section propose potential solutions for future investigation.

8.2.1 Algorithmic Innovations

In our attempt to parallelise algorithms for mutable irregular data we used four auxiliary techniques, namely graph colouring, the deferred-updates mechanism, atomic-based worklist creation and work-stealing using POSIX interrupts. Some of them seem to be optimal, while others could be improved.

Graph colouring Our improved optimistic method seems to be nearly optimal both in terms of execution speed, scaling very strongly even to hundreds of threads, and number of colours, using the same amount as the serial greedy algorithm (which has been shown to produce near-optimal colouring, as was mentioned in Chapter 5). We do not believe that this method can be improved any further.

Atomic-based worklist creation This approach to parallel worklists proved to be very fast, eliminating the need for thread synchronisation and subsequent global reduction on the number of elements each thread needs to push back into the list. It allowed us to have *nowait* for-loops, *i.e.* OpenMP parallel loops where threads do not synchronise at the end of the loop. Considering that thread barriers is one of the dominant factors limiting scalability of PRAgMaTIc, eliminating some of them is important. We cannot think of a more efficient way of creating worklists in parallel.

Deferred-updates mechanism This strategy proved to be key in achieving high performance while keeping the code race-free. Overhead of the mechanism itself is negligible, never showing up in our performance profiling tools. However, in order to use this strategy it is necessary to introduce additional thread synchronisation. All threads must finish execution of an adaptive kernel, synchronise, commit the updates and then synchronise again before proceeding to the next independent set. This results in two thread barriers per independent set and $b_{total} = 2 \times n_{sets}$ barriers per sweep of the adaptive algorithm.

Interrupt-Driven Work-Sharing scheduler The IDWS scheduler was shown to perform very well, achieving optimal load balance (since it implements range-stealing) while mitigating the negative impact on performance when choosing victims randomly. However, profiling the scheduler revealed a prevalent hotspot: waiting for an answer from the victim. We believe that if waiting time is minimised, performance of IDWS will be vastly improved, possibly making the scheduler outperform its main competitor, OpenMP *guided*, in those benchmarks where the latter is currently faster.

8.2.2 PRAgMaTIc

As was discussed in Section 6.5, PRAgMaTIc is fast and scalable, yet its performance can be improved. Two dominant factors were spotted which limit parallel efficiency: thread synchronisation and bandwidth saturation. Admittedly, in our first attempt to parallelise mesh adaptivity we neglected data locality issues. Optimal data reuse is hard when working on irregular data for reasons explained in Section 1.2. However, there is room for improvement.

8.3 Future Work

Reflection on our work from the previous section highlights the main aspects onto which we should shift our focus for future work.

Regarding locality issues in PRAgMaTIc, an immediate solution to this problem could be merging the four adaptive algorithms into a unified super-algorithm. Looking at how those algorithms were parallelised, we can see a common pattern: they process the mesh in batches of independent sets of vertices (with the exception of refinement). Every vertex defines a mesh cavity consisting of all adjacent vertices and elements. Consequently, independent vertices define independent cavities, i.e. local mesh patches on which a thread can perform any operation without worrying about races (with the exception of updating adjacency lists for vertices on the cavity boundary). It is a form of implicit, on-the-fly mesh partitioning. Following this approach, cavity-related data is loaded into the cache once and then all 4 algorithms are applied to the cavity, making effective reuse of what is already in the cache.

A side benefit of the unified super-algorithm is that, subsequently, thread barriers, mesh colourings and committing of deferred operations pertaining to each algorithm are also merged, minimising the overhead of those auxiliary parts of mesh adaptivity. Especially for thread synchronisation, it can be expected that barriers will be reduced by a factor of ≈ 3. However, it must be pointed out that applying adaptive operations using this unified approach effectively constitutes a different mesh adaptation algorithm. We will have to investigate potential compromises (if any) to the resulting mesh quality.

In Section 4.4 we argued that the deferred-updates strategy is a fast alternative to a 2-distance colouring of the graph, which is more time-consuming than a simple 1-colouring and possibly results in more colours being used, effectively limiting the exposed parallelism. In our attempt to minimise thread synchronisation, it is worth exploring whether a 2-distance colouring would eventually be a better approach, since using it will eliminate one thread barrier per independent set, leaving us with $(b_{total})^{2-distance} = 1 \times (n_{sets})^{2-distance}$ barriers per sweep of the adaptive algorithm. This hypothesis will be verified if the 2-distance colouring uses less than double the colours of the 1-distance colouring, so that $(b_{total})^{2-distance} < (b_{total})^{1-distance}$.

Two main points of focus for further work on loop scheduling should be data locality and efficiency of work-sharing. Work on locality issues has been published by several groups ([85, 52, 98, 1]). Data locality has been neglected in this work both for IDWS and PRAgMaTIc; however, it will be quite important in the context of the unified adaptive algorithm. In terms of efficiency, it is worth exploring ways to minimise the thief's waiting time. Moving toward this direction, it could be worth taking a step back and reconsidering use of atomics instead of signals. The main argument against atomics was that every worker would have to constantly access its upper bound atomically, since a thief may have modified it, which is quite expensive. Performance of atomic operations is expected to be improved in the latest and future CPU architectures (*e.g.* on Intel®Haswell an atomic read will be just a regular load from L1 most of the time and fetching the updated value from elsewhere will be necessary only if some other thread has modified it). Finally, Adnan and Sato have presented interesting ideas on efficient work-stealing strategies [3], some of which could be applicable to our work-sharing scheduler.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and RobertD. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. *SIGPLAN Not.*, 48(8):219–228, February 2013.

[3] Adnan and M. Sato. Efficient work-stealing strategies for fine-grain task parallelism. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 577–583, May 2011.

[4] A Agouzal, K Lipnikov, and Yu Vassilevski. Adaptive generation of quasi-optimal tetrahedral meshes. *East west journal of numerical mathematics*, 7(4):223–244, 1999.

[5] F Alauzet. Size gradation control of anisotropic meshes. *Finite Elements in Analysis and Design*, 46(1):181–202, 2010.

[6] Frédéric Alauzet, Xiangrong Li, E Seegyoung Seol, and Mark S Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, 2006.

[7] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms, 1995.

[8] N. Amenta, M. Bern, and 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.

[9] Thomas Apel, Sergei Grosman, Peter K. Jimack, and Arnd Meyer. A new methodology for anisotropic mesh refinement based upon error gradients. *Appl. Numer. Math.*, 50(3-4):329–341, September 2004.

[10] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711–712, 09 1976.

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

[12] 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.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pages 207–216, 1995.

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[15] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[16] Bronis R. de Supinski. Progress on OpenMP Specifications. Presented at Supercomputing 2012 Salt Lake City, `http://openmp.org/wp/presos/SC12/SC12_State_of_LC.2.pdf`.

[17] Gustavo C Buscaglia and Enzo A Dari. Anisotropic mesh optimization and its application in adaptivity. *International Journal for Numerical Methods in Engineering*, 40(22):4119–4136, 1997.

[18] S. Canann, J. Tristano, and M. Staten. An approach to combined Laplacian and optimization-based smoothing for triangular,

quadrilateral, and quad-dominant meshes. In *Proceedings, 7th International Meshing Roundtable*, pages 479—494, 1998.

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

[20] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10–11):576–594, 2012.

[21] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM COMPUTING SURVEYS*, 38(1):2, 2006.

[22] Long Chen, Pengtao Sun, and Jinchao Xu. Optimal anisotropic meshes for minimizing interpolation errors in l^{p}-norm. *Mathematics of Computation*, 76(257):179–204, 2007.

[23] Jonathan Cohen and Patrice Castonguay. Efficient Graph Matching and Coloring on the GPU. `http://on-demand.gputechconf.com/gtc/2012/presentations/S0332-Efficient-Graph-Matching-and-Coloring-on-GPUs.pdf`.

[24] T. Coleman and J. Moré. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[25] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Intel Corporation. Intel®Xeon® Processor E5 v3 Family. `http://ark.intel.com/products/family/78583/Intel-Xeon-Processor-E5-v3-Family`.

[27] Intel Corporation. Intel®Xeon Phi™ Product Family. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[28] Thierry Coupez, Hugues Digonnet, and Richard Ducloux. Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153–175, 2000.

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

[30] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, December 2011.

[31] HL De Cougny and Mark S Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[32] Edsger W. Dijkstra. The origin of concurrent programming. In Per Brinch Hansen, editor, *The origin of concurrent programming*, chapter Cooperating Sequential Processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

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

[34] Eduard Ayguadé. OpenMP in the Petascale Era: Does OpenMP need a more powerfulset of features for tasks?. Presented at Supercomputing 2011 Seattle - OpenMP BOF, `http://openmp.org/wp/presos/StarSs%40OpenMPBOF.pdf`.

[35] Guillaume Fertin, Emmanuel Godard, and André Raspaud. Acyclic and k-distance coloring of the grid. *Inf. Process. Lett.*, 87(1):51–58, July 2003.

[36] D. A. Field. Laplacian smoothing and Delaunay triangulations. *Communications in Applied Numerical Methods*, 4:709—712, 1988.

[37] L. Freitag, M. Jones, and P. Plassmann. An efficient parallel algorithm for mesh smoothing. In *Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories*, pages 47–58. Citeseer, 1995.

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

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

[40] Lori F. Freitag, Mark T. Jones, and Paul E. Plassmann. The Scalability Of Mesh Improvement Algorithms. In *IMA VOLUMES IN MATHEMATICS AND ITS APPLICATIONS*, pages 185–212. Springer-Verlag, 1998.

[41] PASCAL-JEAN Frey and Frédéric Alauzet. Anisotropic mesh adaptation for cfd computations. *Computer methods in applied mechanics and engineering*, 194(48):5068–5082, 2005.

[42] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[43] Fujitsu. FUJITSU Supercomputer PRIMEHPC FX10. `http://www.fujitsu.com/global/services/solutions/tc/hpc/products/primehpc/`.

[44] Fujitsu. K computer. `http://www.fujitsu.com/global/about/businesspolicy/tech/k/`.

[45] Fujitsu. OpenMP Application Program Interface, Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

[46] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[47] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.

[48] Gerard J. Gorman, Georgios Rokos, James Southern, and Paul H. J. Kelly. Thread-parallel anisotropic mesh adaptation. *Accepted for Publication in Proceedings of the 4th Tetrahedron Workshop on Grid Generation for Numerical Computations*, 2014.

[49] GJ Gorman, J Southern, PE Farrell, MD Piggott, G Rokos, and PHJ Kelly. Hybrid OpenMP/MPI anisotropic mesh smoothing. *Procedia Computer Science*, 9:1513–1522, 2012.

[50] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation : P-completeness theory.* Oxford University Press, New York, 1995.

[51] ISS Group. Galois. `http://iss.ices.utexas.edu/?p=projects/galois`.

[52] Yi Guo, Jisheng Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[53] R.A Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A Blumrich, R.W. Wisniewski, A Gara, G.L.-T. Chiu, P.A Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[54] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC ’02, pages 280–289, New York, NY, USA, 2002. ACM.

[55] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. *SIGPLAN Not.*, 47(4):349–362, March 2012.

[56] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, June 2000.

[57] Cray Inc. Cray XE6 Supercomputer. `http://www.cray.com/Products/Computing/XE/XE6.aspx`.

[58] J. Jeffers and J. Reinders. *Intel® Xeon Phi^{TM} Coprocessor High-performance Programming.* Morgan Kaufmann. Elsevier Science & Technology Books, 2013.

[59] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[60] Peter Johnsen, Mark Straka, Melvyn Shapiro, Alan Norton, and Thomas Galarneau. Petascale wrf simulation of hurricane sandy deployment of ncsa's cray xe6 blue waters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 63:1–63:7, New York, NY, USA, 2013. ACM.

[61] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. SCI. COMPUT*, 14:654–669, 1992.

[62] Mark T. Jones and Paul E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Comput.*, 20(5):753–773, May 1994.

[63] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[64] P.M. 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):401–420, 2000.

[65] P.M. 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):1165–1185, 2000.

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

[67] S. Korotov. Error control in terms of linear functionals based on gradient averaging techniques. *Computing Letters*, 3(1):35–44, 2007.

[68] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, pages 65–76. IEEE, 2009.

[69] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, 1949*, pages 141–146, Cambridge, Mass., 1951. Harvard University Press.

[70] C.Y. Lepage, A. St-Cyr, and W.G. Habashi. Mpi parallelization of unstructured mesh adaptation. In Clinton Groth and DavidW. Zingg, editors, *Computational Fluid Dynamics 2004*, pages 727–732. Springer Berlin Heidelberg, 2006.

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

[72] Konstantin Lipnikov and Yuri Vassilevski. Parallel adaptive solution of 3d boundary value problems by hessian recovery. *Computer Methods in Applied Mechanics and Engineering*, 192(1112):1495 – 1513, 2003.

[73] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[74] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new

framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[75] M Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[76] ANDREW LUMSDAINE, DOUGLAS GREGOR, BRUCE HENDRICKSON, and JONATHAN BERRY. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[77] Kamesh Madduri. Snap (small-world network analysis and partitioning) framework. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1832–1837. Springer US, 2011.

[78] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[79] Fredrik Manne. A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices. In *Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, PARA '98, pages 332–336, London, UK, UK, 1998. Springer-Verlag.

[80] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 104–113, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[81] David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 109 – 122. Academic Press, 1972.

[82] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[83] K. Morgan, J. Peraire, J. Peiro, and O. Hassan. The computation of three-dimensional flows using unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 87(23):335 – 352, 1991.

[84] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM.

[85] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[86] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008.

[87] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, July 2011.

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

[89] C. C. Pain, A. P. Umpleby, C. R. E. de Oliveira, and 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):3771 – 3796, 2001.

[90] CC Pain, MD Piggott, AJH Goddard, F. Fang, GJ Gorman, DP Marshall, MD Eaton, PW Power, and CRE De Oliveira. Three-dimensional unstructured mesh ocean modelling. *Ocean Modelling*, 10(1-2):5–33, 2005.

[91] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, October 1988.

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

[93] 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.

[94] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[95] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[96] R. Rabenseifner, G. Hager, and 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*, pages 427–436. IEEE, 2009.

[97] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, Berkely, CA, USA, 1st edition, 2013.

[98] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[99] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engg.*, 15(2):66–71, March 2013.

[100] Georgios Rokos and Gerard Gorman. PRAgMaTIc - Parallel Anisotropic Adaptive Mesh Toolkit. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 143–144. Springer Berlin Heidelberg, 2013.

[101] Georgios Rokos, Gerard Gorman, and PaulH.J. Kelly. Accelerating anisotropic mesh adaptivity on nvidias cuda using texture interpolation. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 387–398. Springer Berlin Heidelberg, 2011.

[102] Y. Saad. Ilum: A multi-elimination ilu preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 17(4):830–847, 1996.

[103] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. *SIGARCH Comput. Archit. News*, 38(1):311–322, March 2010.

[104] R. B. Simpson. Anisotropic mesh transformations and optimal error control. *Appl. Numer. Math.*, 14(1-3):183–198, April 1994.

[105] J. Southern, GJ Gorman, MD Piggott, and PE Farrell. Parallel anisotropic mesh adaptivity with dynamic load balancing for cardiac electrophysiology. *Journal of Computational Science*, 2011.

[106] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Park, Maryland, July 25-27, 2002.

[107] Srikant Subramaniam and Derek L. Eager. Affinity scheduling of unbalanced workloads. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 214–226, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[108] A Tam, D Ait-Ali-Yahia, MP Robichaud, M Moore, V Kozel, and WG Habashi. Anisotropic mesh adaptation for 3d flows on structured and unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 189(4):1205–1230, 2000.

[109] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In

Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *ISAAC (2)*, volume 6507 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2010.

[110] The MPI Forum. MPI: A Message Passing Interface, 1993.

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

[112] Eric W. Weisstein. Multiplicative Order. `http://mathworld.wolfram.com/MultiplicativeOrder.html`.

[113] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[114] Keiji Yamamoto, Atsuya Uno, Hitoshi Murai, Toshiyuki Tsukamoto, Fumiyoshi Shoji, Shuji Matsui, Ryuichi Sekizawa, Fumichika Sueyasu, Hiroshi Uchiyama, Mitsuo Okamoto, Nobuo Ohgushi, Katsutoshi Takashina, Daisuke Wakabayashi, Yuki Taguchi, and Mitsuo Yokokawa. The k computer operations: Experiences and statistics. *Procedia Computer Science*, 29(0):576 – 585, 2014. 2014 International Conference on Computational Science.

[115] O. C. Zienkiewicz and R. L. Taylor. *Finite Element Method*, volume 1. Butterworth-Heinemann, fifth edition, 2000.

[116] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method Set.* Butterworth-Heinemann, sixth edition, 2005.