CRANFIELD UNIVERSITY

ANTONIO PEDRO ARAUJO FRAGA

THE EFFECT OF MESH PARTITIONING QUALITY ON THE PERFORMANCE OF A SCIENTIFIC APPLICATION IN AN HPC ENVIRONMENT

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Software Engineering for Technical Computing

MSc
Academic Year: 2017–2018

Supervisor: Dr I. Moulitsas
August 2018

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Software Engineering for Technical Computing

MSc

Academic Year: 2017–2018

ANTONIO PEDRO ARAUJO FRAGA

The effect of mesh partitioning quality on the performance of a scientific application in an HPC environment

Supervisor: Dr I. Moulitsas
August 2018

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.

Abstract

The need of fast and reliable methods to solve large linear systems of equations is growing rapidly. Because this is a challenging problem, several techniques have been developed in order to solve it accurately and efficiently. Geometric Multigrid methods are being used to solve these problems, as they accelerate the convergence to a solution. With these methods, it is possible to use a coarser grid as an input, reducing the problem domain and thus reducing the computational cost.

The focus of this thesis is the development of new algorithms to generate a sequence of coarser grids from the original grid. By treating this problem as a minimization problem, one can attempt to optimize the overall grid quality by choosing how to merge elements. In order to evaluate our algorithms, we are going define how to quantify the overall grid quality, and therefore analyse the grids obtained by them. We are also going to use the multilevel grid construction paradigm, which is known to be adequate to solve similar problems.

Such construction can be done in parallel, by adding a small overhead and not sacrificing the quality produced by our multilevel constructor. Hence, we can achieve a high level of concurrency.

Contents

List of Figures

List of Tables

List of Abbreviations

SATM	School of Aerospace, Technology and Manufacturing
HPC	High Performance Computing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
API	Application Programming Interface
OpenMP	Open Multi-Processing
MPI	Message Passing Interface
CUDA	Compute Unified Device Architecture
SPMD	Single Program, Multiple Data
MIMD	Multiple Instruction, Multiple Data
SIMD	Single Instruction, Multiple Data
PU	Processing Unit

Acknowledgements

I would like to express my gratitude to my family, who always supported me with my choices and my education. I have to thank my father for the values and good manners that he persistently tried to pass me. I have to thank my mother for the love and empathy in my everyday life. I would like to thank my brother for all the advises in times of stubbornness, making me a better brother, and a better man.

Secondly, I want to thank my supervisor, Dr. Irene Moulitsas, for her teaching, her patience and support with my academic and future professional life. With special thanks to Soren Rasmussen, for several technical advices and for, the always, available help.

I'm grateful to everyone who contributed for my interest in Computer Science, Math, Physics and Logic, that in some extent helped me to write this thesis.

Chapter 1

Introduction

The need of fast and reliable methods to solve large linear systems of equations is growing rapidly. The Aerospace industry, for instance, is focused on predicting the fluids behaviour around the several aircraft components. This simulation is possible by solving equations called *Euler* and *Navier-Stokes* [30, 15]. These systems play an important role on weather and climate forecasting as well. The results can be influenced by many observations around the Earth surface, and in case of Met Office in the UK, they receive millions of those observations [3].

Because this is a challenging problem, several methods have been developed in order to solve it accurately and efficiently. These techniques have different benefits and drawbacks, and there is no standard method that produces significantly good results on both metrics. Accuracy presents, in most cases, proportionally inverse improvements when comparing to efficiency.

One can split methods for solving linear systems of equations into two categories, **direct** and **iterative**. Direct methods are able to obtain a solution by performing a set of operations or following a formula. The solution will not be ready until all operations are complete. *LU decomposition* is an example of a direct method. Iterative methods, on the other hand, converge to an approximated solution. One should establish a limit of iterations, where hopefully, a sufficiently accurate solution is found [1].

Nevertheless, some problems are very large, and is very time consuming to solve them. Thus, new methods have to be introduced. Methods that will speed up the solution process, but still obtain a reasonable solution.

1.1 Multigrid Methods

This problem can be solved by making use of multigrid methods. Given a system of equations $A.x = b$ where $A \in R^{n\times n}, x, b \in R^n$ and x is unknown, it is faster to solve a problem with a lower number of unknowns. Hence, one can try to decrease the domain size, by generating a smaller (approximated) **A** matrix. Hence, the obtained solution, being a smaller matrix, can be used as a starting point of an iterative method again. The methods to produce a coarser grid can be categorized as Algebraic and Geometric.

In an Algebraic approach all unknowns are treated separately, and by declaring a system $A.x = b$,

$$\begin{bmatrix} A_{(0,0)} & \cdots & A_{(0,n)} \\ \vdots & \ddots & \vdots \\ A_{(n,0)} & \cdots & A_{(n,n)} \end{bmatrix} \begin{bmatrix} x_{(0)} \\ \vdots \\ x_{(n)} \end{bmatrix} = \begin{bmatrix} b_{(0)} \\ \vdots \\ b_{(n)} \end{bmatrix}$$

, where n indicates the number of unknowns in the given system, an interpolation between the several entries is made. To be specific, when having an unknown variable on index i, only variables related with the *i-th* unknown are interpolated [11].

In Geometric multigrid approaches, a hierarchy of grids is created, by successively decreasing the number of points recursively. In this method, coarser and coarser grids are generated by grouping a set of points into one single point in each coarsening step. These agglomerated points can be defined as control volumes [24].

Future work will be based on a previously developed software, MGridGen/ParMGrid-Gen, that makes use of a Geometric Approach. It is important to mention that there is no single method to generate a set of coarse grids, nor to choose which points should be put together. This software, however, treats the choice of points to be put together as an optimization problem. The sequence of coarse grids must generate well shaped control volumes. Therefore, a minimization metric called **Aspect Ratio** was introduced, that defines what means to have a "*well shaped*" control volume,

$$A = \frac{l^2}{S}$$

, where A is defined to be the aspect ratio for two dimensional matrices. l is the circumferential length and S is the area of the control volume. Hence, for three-dimensional grids, one can defined this metric as,

$$A = \frac{S^{\frac{3}{2}}}{V}$$

, where S is the surface area and V is the volume of the whole control volume. As mentioned, this metric should be minimized in order to generate control volumes that are as close of having a circular / spherical shape as possible [27]. The reason why a minimization of this value will converge to such shape can be explained by stating that a circle has a smaller circular length that any other shape with the same area. Similarly, a sphere has a smaller surface area than any other solid with the same volume. By taking a two-dimensional example, represented on figure 1.1, containing a circle and an hexagon with the same area,

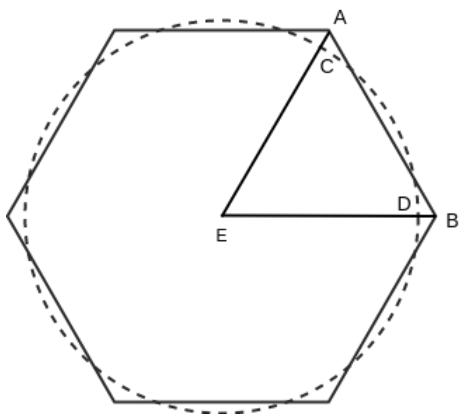

Figure 1.1: Circle and hexagon with the same area.

the perimeter of the hexagon is 6 times the length of $\overline{AB}$, whereas the perimeter of the circle is 6 times the length of $\overset{\frown}{CD}$. In a similar polygon with **n** equal sides, the length of a side is given by $tan(\frac{\pi}{n})$, but a coincident arc of a circle will have a length of $\frac{\pi}{n}$. Since $n \in \mathbb{N}$, one can state that $tan(\frac{\pi}{n}) > \frac{\pi}{n}$, and therefore the perimeter of such polygon is greater than the perimeter of a circle with equal area. The value will be even larger when considering irregular polygons. Thus, in order to converge to a circle, one would have to decrease the perimeter of the polygon, or, in other hand, increase its area. And by analysing both two- and tree-dimensional formulas of the **Aspect Ratio**, one can minimize its value by applying the exact same changes.

Having the circular length as a power of 2, will make the Aspect Ratio to converge to a constant value without any variables in our final result. In order to follow the same idea in three-dimensions, we declare the surface area as a power of $\frac{3}{2}$. Such optimization, having circular / spherical control volumes, tends to result in a solution obtained by fewer iterations.

1.2 Problem Complexity

Such optimization problem can not be solved quickly. Problems are often categorized by its solution **time complexity**. This term defines on how will the execution time increase as the problem domain is increasing. Hence, two main categories were created. The problems in the first category are defined as problems that can already be solved in **polynomial** time, or **P** problems. This means that its solution has a number of steps which is bounded by a **polynomial** function of **n**, where **n** is defined to be the length of the problem. In the second category we include problems where its solution can be guessed and verified in polynomial time, called **Nondeterministic-polynomial** problems or **NP**-Problems, where *nondeterministic* means that there is no guide or rule to make a guess. This category contains problems to which a solution with a *polynomial* time complexity was not found yet.

Nevertheless, two more controversial categories can be defined. A problem is said to be NP-hard if its solution can be modified to solve any NP-Problem. Thus, having an NP-Problem that is NP-hard would originate in an NP-complete problem. This area is quite controversial, since finding an efficient solution for an NP-complete problem will end up by meaning that an efficient solution was found for every NP-Problems [17, 4]. The previous statements can be visually represented in figure 1.2.

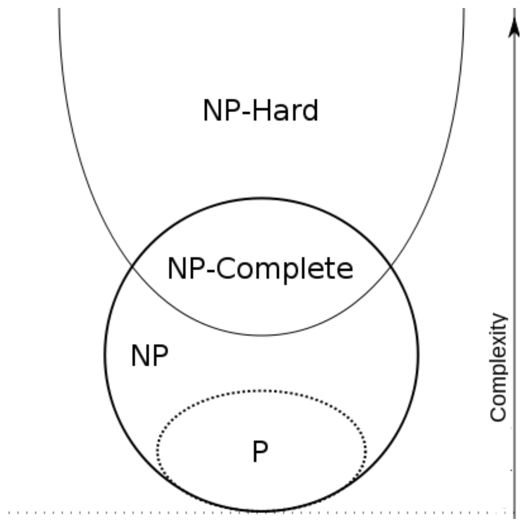

Figure 1.2: Time complexities visual representation.

The referred optimization problem is given as an **NP**-Problem, where for now, *guesses* have to be made in order to obtain a solution in polynomial time. There are several ways of solving these kind of problems, and one would have to come up with a method that would obtain an approximated solution, similarly to what happens with iterative methods. A different approach is to try to predict what method would be capable of solving a reasonable amount of cases, in other words, try to define a method that often produces good results. Such method is defined as an **heuristic** method. But because heuristic methods are shown to often present good results, it doesn't mean that results with a good quality will always be presented.

1.3 Parallelisation

The importance of parallel computing has been recognized for the last years. This paradigm has made an enormous impact in a set of areas like *computational simulations*, *data mining* and *transaction processing*. Rather than having a single-core (**monolithic**) processor, new integrated circuits called multi-core processors were introduced. In these systems, multiple instances of a program can run at the same time [28].

1.3.1 Hardware

Several architectures were developed in order to attend the needs of such model. The multi-core **C**entral **P**rocess **U**nit and the **G**raphics **P**rocess **U**nit are two examples of them. These processors have different characteristics that makes them suitable to deal with different situations. The number of cores in a CPU is often smaller when compared to the number of cores present in a GPU device. Notice that one can also state that the number of cores in a common home computer is significantly smaller when compared to a high end computer. As the name suggests, a GPU was initially built to perform computer graphics calculations. Nevertheless, the computation power of a CPU core is often higher when compared to a GPU core, but the level of parallelism is lower. Most of the tasks can be processed by the CPU, specially when it is important to maximize the performance of a single task.

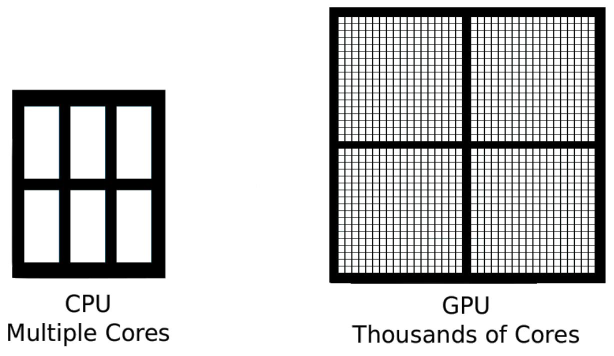

Figure 1.3: Number of cores comparison in a CPU and GPU.

The GPU plays an important role when in need of executing hundreds of smaller tasks. The reason why we call them small, is because of the GPU cores memory limitations. Thereby, having such a large number of cores would mean that it is possible to have a larger number of execution flows at the same time. The comparison of cores between a CPU and GPU can be represented in figure 1.3.

1.3.2 Software

Parallel software had to be developed in order to take advantage of these systems. Software developers started to build programs that exploited shared- and distributed-memory architectures. Among other types of techniques, Flynn classified multiprocessors as *M*ultiple **I**nstructions, **M**ultiple **D**ata (*MIMD*), and **S**ingle **I**nstruction, **M**ultiple **D**ata (*SIMD*). *MIMD* is characterized by having multiple execution flows operating on different data, whereas in *SIMD*, the parallel units share the same instructions on multiple data [10]. The **S**ingle **P**rogram, **M**ultiple **D**ata (*SPMD*) paradigm is included in the *MIMD* category, executing the same program with independent execution flows. *MIMD* is often considered to be the dominant style of parallel programming. By following it, in a shared-memory paradigm, tasks are carried by several *threads* running in the same process. W when running distributed-memory programs, *processes* are the ones carrying out tasks.

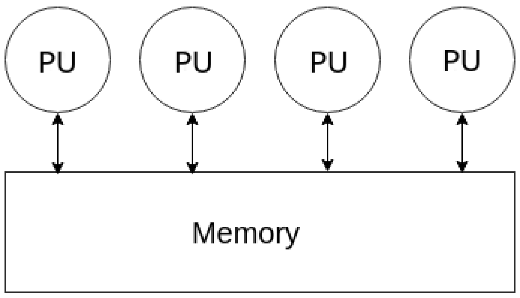

Figure 1.4: An example of a shared memory architecture.

When dealing with a shared-memory architecture, several threads can read/write shared variables at the same time, whereas private variables can only be read/written within a private execution flow. The fact of having variables that can be modified/read at the same time can create unexpected results. In order to solve this problem, several synchronization mechanisms had to be introduced. These issues can be solved by insuring **mutually exclusive** access to critical sections. Figure 1.4 represents an example of such system, where **PU** stands for **P**rocessing **U**nit.

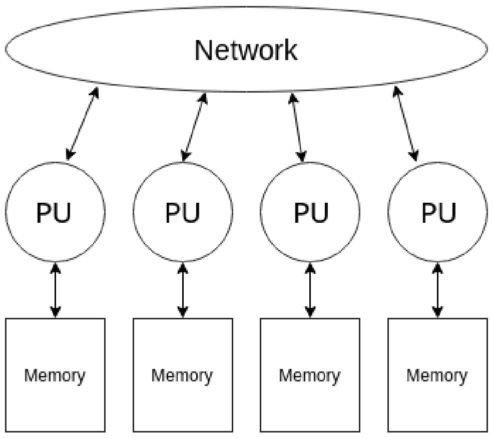

Figure 1.5: An example of a distributed memory architecture.

On the other hand, in a distributed-memory architecture, processes usually don't share memory among each other. Processes should communicate in order to exchange information. One can achieve such communication with commonly used APIs, like message-passing or partitioned global address space. This paradigm allows multiple processes to be running on independent CPUs within independent systems. The system in figure 1.5 has distributed-memory architectu

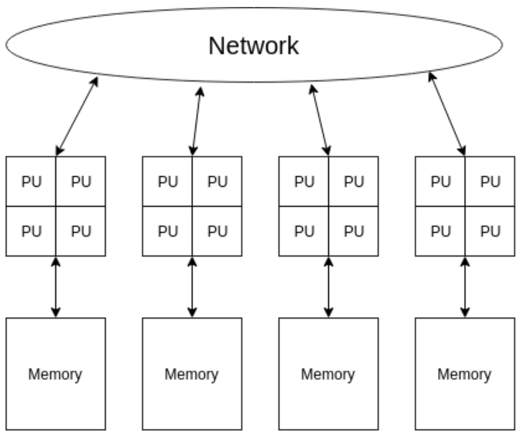

Figure 1.6: An example of a hybrid architecture.

By keeping in mind the last model, each node that is forming the system has a shared memory architecture with one or more multicore processors [28]. If this is the case, processes can also fork multiple threads on each node. Such systems are defined as **hybrid**, and they can be represented in figure 1.6.

A considerable number of parallel implementations were developed during the last decades. Their goal is to offer a higher-level API that implements the same functionalities. OpenMP and MPI are technologies that make use of the shared- and distributed-memory architectures respectively. CUDA is also a parallel computing platform that was created by Nvidia. The last platform is a software layer for the execution of compute kernel in a GPU. Notice that CUDA is limited to Nvidia devices, but different technologies can be used to achieve the same purpose with devices from other vendors [23, 28].

Chapter 2

Literature Review

Several papers will be summarized and evaluated. The research will appear in chronological order. The level of accuracy and relevance will be discussed.

2.1 Path, Trees and Flowers

The assignment problem is known as maximum weighted matching problem for bipartite graphs. In 1965, Jack Edmonds, proposed an algorithm which was capable of finding a maximal matching in a non-bipartite graph *G = (V, E)* in polynomial time. The algorithm could find a matching *M*, such that each vertex is incident with at most one edge in *M* and $|M|$ is maximized.

The main idea of this algorithm was to add alternating edges of a path to a Match until a cycle is found. It has a particular characteristic of being able to deal with augmented paths with an odd-length cycle, which was contracted to a single vertex, forming a new interior graph, and a new contracted graph. The new augmented path had similarities with a flower, where the cycle could be seen as the blossom, and the remaining extended path was apparent to a stem. Several expansions would create a tree.

This algorithm had a time complexity of $O(|E||V|^2)$, where $|E|$ represents the total number of edges, and $|V|$ represents the total number of vertexes [7].

2.2 Maximum matching and a polyhedron with 0,1-vertices

Edmonds, adpated the initial algorithm to find a maximum or minimum weighted matching, by implementing a variant of the Hungarian Method on bipartite cases. Therefore, the new algorithm consisted of the maximum cardinality algorithm described in section 2.1 with small modifications [6]. The new algorithm was capable of producing a matching M, such as M includes the minimum or maximum total weight.

2.3 Moore's law

For the past years, Moore's law has been served as a guide of the integrated circuits evolution. Initially being an observation, Moore could predict trends in the premature times of integrated circuits. Such predictions would claim that the number of transistors that can be inexpensively placed on an integrated circuit would double roughly every 18 months. It is worth to mention that such prognosis was strongly linked to the cost reduction of integrated systems components. Notice that several characteristics like memory capacity and computational power are connected to this claim. By adding new features to the integrated circuit, more transistors had to be included, leading to larger chips. Such additions were being followed by a compaction of the original design, originating in a more efficient and economical circuit. Thus, with such process, integrated circuits were being shrunk over time [33]. This trend was initially described in 1965, and Moore predicted that it would evolve in the same way ”for at least ten years” [26]. Such prediction was later confirmed. As the years passed by, Moore's law was being declared accurate and long-lasting. After more than 50 years, chipmakers are considering even more seriously different strategies to advance their platforms. As this law is starting to wind down. Alternatives like quantum, neuromorphic and optical processors are being explored [8].

2.4 Amdahl's law

As parallel computing was being introduced, methods for speeding the execution time were being explored. Chips with multiple processors were being announced, and the computational power was evolving by repeatedly doubling the number of cores per chip. The development of parallelized applications was increasing, and the need of studying parallel execution times was increasing. By declaring the term *Speedup* as the original execution time divided by the enhanced execution time, one can formulate,

$$Speedup = \frac{OriginalExecTime}{EnhancedExecTime}$$

Amdahl introduced the idea of taking into account the fact that a portion of the program might not be parallelised. This idea includes a fixed workload, and a workload which execution time can be enhanced. The theoretical speedup in latency of execution of a given task could be obtained. This law can be formulated as follows,

$$S_{latency} = \frac{1}{r_s + \frac{r_p}{n}}$$

, with r_p being the ratio of the parallel portion in one program. Similarly, r_s was declared as the sequential portion. The variable n represents the number of processors executing the program in parallel. Such statements can introduce one extra condition $r_p + r_s = 1$ [2, 29].

2.5 Gustafson's law

Gustafson re-evaluated Amdahl's law, and claimed that the problem size scales with the number of processors. In other words, instead of fixing the problem size, a parallel program would allow us to increase it. By using r_p and r_s to represent the serial and parallel time in a parallel system, then it would be necessary $r_s + r_p \times n$ execution time. Formulating,

$$S_{scaled} = r_s + r_p \times n$$

To conclude, this law states that by increasing the the problem size, one can retain scalability with respect to the number of processors [13]. Nevertheless, these speedups were considered optimistic, as they don't take into account bottlenecks like communication costs or memory bound limitations.

2.6 A linear-time heuristic for improving network partitions

This paper introduces an idea of what is called as *FM algorithm*. Where given two blocks *(A, B)*, one cell can be removed from one block to another in order to minimize the number of cuts to be made within a graph. The cell to be chosen is based on its effect on the size of the next set of unmatched cells [9].

2.7 A multilevel algorithm for partitioning graphs

This paper is among the first work in this area. It introduces the idea of having a NP-complete problem, and that heuristics should be used in order to obtain an approximated solution.

A coarsening technique is introduced by finding a maximal matching with either a *depth-first search* or a randomized algorithm. It explains the concept of *vertex-weight*.

It also explains that uncoarsed graphs have more levels of freedom than smaller coarsed graphs, therefore, a *local refinement* scheme can be applied. This refinement scheme is based on an algorithm developed by Kerninghan and Lin, usually called **KL**. In this algorithm the gain of moving vertices from one control point to a different control volume is analysed during the refinement process. Some size constraints are established in order to obtain balance control volumes [14].

2.8 Multilevel k-way partitioning scheme for irregular graphs

This document clarifies the k-way partitioning scheme. Specifically, the uncoarsening phase. This is done by projecting the graph with *m* partitions (G_m) back to the original graph G_1. Thus, the sequence of graphs in this phase is given as G_m, G_{m-1}, G_{m-2} ... G_1. This idea can be graphically represented in figure 2.1.

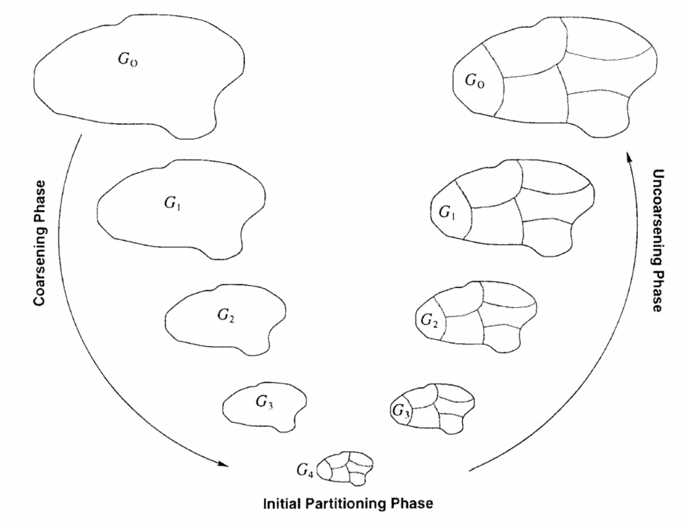

Figure 2.1: An example of a multilevel k-way partitioning.

As this propagation is being made, partitions are gaining more vertexes. For instance when declaring a level of partition G_n, it is know that level G_{n-1} has more degrees of freedom. This means that is possible to improve the partitioning to further decrease the *edge-cut* value. It is performed a *backtracking-like* procedure, where subsets of vertexes are moved from one control-point to another.

The results obtained show that it is not guaranteed to improve the quality of a given solution at this phase, but is something that is definitely possible.

Some algorithms regarding this phase were discussed, like *Greedy Refinement* and *Global KernighanLin Refinement* [21].

2.9 A fast and highly quality multilevel scheme for partitioning irregular graphs

This paper introduces a study of several algorithms to be applied in the coarsening phase. There are at least two algorithms discussed that are relevant to the future work of this project. They are called *Random Matching* and *Heavy-Edge Matching*.

The first algorithm consists in choosing vertexes in a randomized approach. One can declare such vertex as *u*. If *u* exists, then the algorithm should select one of its unmatched adjacent vertices. One can name the second vertex as *v*. Notice that this relation corresponds to an edge (*u, v*). If there are no unmatched adjacent vertices, then vertex *u* remains unmatched.

The second algorithm can be declared as a greedy approach. Similarly to what happens in the previous algorithm, the approach is to choose an unmatched vertex *u* randomly. But the method won't randomly choose a vertex *v*. It will instead, choose the adjacent vertex that would minimize the weight of the next graph to be analysed. In other words, it will choose the edge that will maximize the weight of the sub-graph that is being coarsen. Thus, the coarser graph decreases its edge-weight with a solution that it seems to be the best at that time.

When comparing the two methods, the coarsening time of *Random Matching* is only up to 4% less that the coarsening time of the *Heavy-Edge Matching* approach. Nevertheless, the uncoarsening time of *Random Matching* can sometimes be, 50% higher than the time measured with *Heavy-Edge Matching* [20].

2.10 Parallel multilevel k-way partitioning for irregular graphs

This paper presents a parallel approach to implement a multilevel k-way partitioning algorithm. It starts by explaining what multilevel k-way partitioning is, followed by a formulation of their approach to solve such problem. They solve the problem by using a graph colouring approach.

In the coarsening phase, is possible to exchange vertices with the same colour between processors. Thus, it is assured that these vertex movements will reduce the edge-cut. The evaluation of which vertices can be properlymoved creates a need of communication and synchronization. Even though that these factors are an execution time bottleneck, some improvements could be observed [19, 12].

2.11 Multilevel algorithms for generating coarse grids for multigrid methods

The software (MGridGen/ParMGridGen) was based on this research, as these approaches are implemented in the software.

This paper focus in the development of algorithms for generating a sequence of coarse grids from the original grid. The algorithms should be capable of solving an optimization problem with serial and parallel approaches using a multilevel paradigm. Existing coarsening algorithms were studied, and several metrics of quality were established. These metrics are related with the optimization problem, more specifically a minimization problem.

These grids can be represented as graphs, where each vertex can correspond to a grid element. The graphs contain enough information for being able to solve the given minimization problem. *Vertex-weight* is an important value, which can be used in the coarsening processes as described in section 2.9.

An uncoarsening phase is included as well, as it is expected to propagate the obtained solution into the original graph. This phase is based on what was described in section 2.8, but the approach is described by visiting vertices in a random order. It will posteriorly evaluate the reduction in the value of a given objective function, in case a vertex is moved to a different control-point. The change occurs whenever this condition is met without violating size constraints.

The parallel approach strategy consists in distributing the mesh among processors, where the difference of elements between each processor should be minimized. Since the processors are solving the problem with the serial algorithm within each domain, the quality of interior elements is significantly good. But because it is not allowed to create fused elements across processors, the elements formed at the domain boundaries may suffer from a bad solution.

The results of both serial and parallel algorithms were discussed. It was seen that the performance of each serial algorithm didn't vary much. There are some results that have significant variances when compared to results obtained with different algorithms. For the parallel algorithm, despite the referred issue, the performance increases as the number of processes (up to 16) increases. The parallelized approach is capable of obtaining fairly ”good” results [27, 18].

2.12 A simple approximation for the Weighted Matching Problem

This research was made public after the paper described on section 2.11 being published. A new approach, *Path Growing Algorithm* was developed. This method is declared to find a maximum weight matching in a given graph. Starting with a path of length **zero**, the algorithm tries to extend the path in a given direction for as long as possible. The path is extended by choosing the heaviest edge, and deleting all other edges in the current vertex. This happens until its no longer possible to extend that path. In that case, a new path is created from a different vertex. In the end all vertices should belong to a given path. Even if the path has length **zero**. This algorithm has a linear time efficiency [5].

2.13 Mesh Partitioning: A Multilevel Ant-Colony-Optimization Algorithm

It was intended to achieve a quality graph portioning by using an *Ant Colony Algorithm*. This algorithm uses a *metaheuristic* and is based on probabilities. While building the solution, they consider heuristic information, which is associated with ant trails that are dynamically changing.

The results show that this algorithm performed well in very small graphs. Several improvements are suggested by the authors.

2.14 Engineering a Scalable High Quality Graph Partitioner

One part of this research was conducted by comparing serial matching algorithms. It was seen that the algorithm described in section 2.12 was capable of achieving better results than *Sorted Heavy Edge Matching* and *Heavy Edge Matching* [16].

2.15 Engineering Multilevel Graph Partitioning Algorithms

This paper contains a collection of algorithms that can be used to solve graph partitioning problems.

Novel local improvement algorithms and global search strategies were transferred from existent linear solvers. The refinement algorithms are based on max-flow min-cut techniques.

By using an algorithm described in section 2.12, this paper confirms that this algorithm achieves ”empirically considerably better results” than *Sorted Heavy Edge Matching*, *Heavy Edge Matching* and *Random Matching*.

Several refinement schemes were introduced, one that moves vertices between control volumes like those referred in section 2.8, and the second is declared as a *two-way local search algorithm*. This method consists on keeping one priority queue per each pair of control volumes being considered. Each priority is defined with the gain of changing a vertex to a different control volume. The strategy to select a block whose node can be moved is defined as *Top-Gain*. Restrictions were added in order to converge to a load balanced scenario. Local improvement schemes were discussed [31].

Chapter 3

Methodologies

3.1 Optimization Problem

Having a metric that is capable of measuring the quality of a control volume, one can define several techniques to measure the overall quality of the entire grid. These techniques were previously defined as **Objective Functions**. If N is defined as the number of control volumes in the coarse grid, one can define the first function as,

$$F1 = \sum_{i=1}^{N} A_i$$

, where A_i is the **Aspect Ratio** of the *i-th* control volume. It is trivial that this function must be minimized, but it has a few limitations. Different control volumes can have different aspect ratios. Having one control volume with a large aspect ratio value would ”penalise” a solution that is considered to have a good solution for most of the other control volumes. Therefore, one control volume with a poor aspect ratio would turn a good solution into a bad one.

The previous function can be modified by giving higher weights to larger control volumes.

$$F2 = \sum_{i=1}^{N} w_i A_i$$

In this function, w_i is the number of elements that were put together to form the *i-th* control volume. Despite of attenuating the problem of the previous function, this issue is not eradicated. Having a few control volumes with poor aspect ratios, would still potentially create a wrong evaluation of the overall grid quality.

One can create a different function **F3**. This function will look at the worst aspect ratio in a particular grid. This value, similar to what happened in the previous functions, would have to been minimized. Thereby, one can define,

$$F3 = max_{i=1}^{N} A_i$$

One more function was created, **F4**. It is defined by dividing the Aspect Ratio of the i-*th* control volume by a limit of what would be an acceptable value. Hence, one can formulate,

$$F4 = \sum_{i=1}^{N} \left(\frac{A_i}{Limit} \right)^2$$

Because the objective function is given as the sum of squared ratios, it will ”penalize” values greater than 1, and ”reward” values less than 1. Nonetheless, a *Limit* has to be established. Hence, the Aspect Ratio of a Square/Cube was proposed as such limit. Notice that both aspect ratios lead to a constant value, by taking the two dimensional case as an example,

$$A = \frac{l^2}{S} \qquad \text{The } \mathbf{A} \text{ of a square of side } m \text{ is given as,}$$

$$A = \frac{(4m)^2}{m^2} \qquad \text{and by factoring out } m^2\text{,}$$

$$A = 4^2 = 16 \qquad \text{hence, we can reformulate,}$$

$$F4 = \sum_{i=1}^{N} \frac{{A_i}^2}{256}$$

The same idea can be applied in the three dimensional case,

$$A = \frac{S^{3/2}}{V}$$ The **A** of a cube of length *m* is given as,

$$A = \frac{(6m^2)^{3/2}}{m^3}$$ that can be translated into,

$$A = \frac{6^{3/2} \times m^3}{m^3}$$ and by simplifying,

$$A = 6^{3/2} = \sqrt{216}$$ therefore, F4 can be reformulated,

$$F4 = \sum_{i=1}^{N} \frac{{A_i}^2}{216}$$

By defining the previously described functions, one can formulate an optimization problem, such as: Having an initial large grid, one can generate a smaller grid with a number of control volumes between *Lmin* and *Lmax*. The generated coarse grid must minimize at least one of the functions, **F1**, **F2**, **F3**, **F4**. The algorithm goal is to first minimize **F3**, and then minimize **F1**, **F2** or **F4** [27].

3.2 Multilevel Coarse Grid Construction

An algorithm capable of solving the minimization problem, described in section 3.1, was previously developed. It is based on the **multilevel** paradigm, a geometric multigrid approach. Points are converged, successively forming smaller grids at each iteration. As discussed in section 1.1, these grids are an approximation of the original problem, forming a sequence of grids $\{G_0, G_1, ..., G_n\}$. Once this process is complete, the obtained grid is continuously optimized. Grid G_n is used to find a finer approximation, G_{n-1}. And the

G_{n-1} grid is further refined on G_{n-2}. By introducing the refinement phase, the solution ends up by propagating to the original grid, G_0. Figure 3.1 represents this approach with an example of a simple multilevel construction.

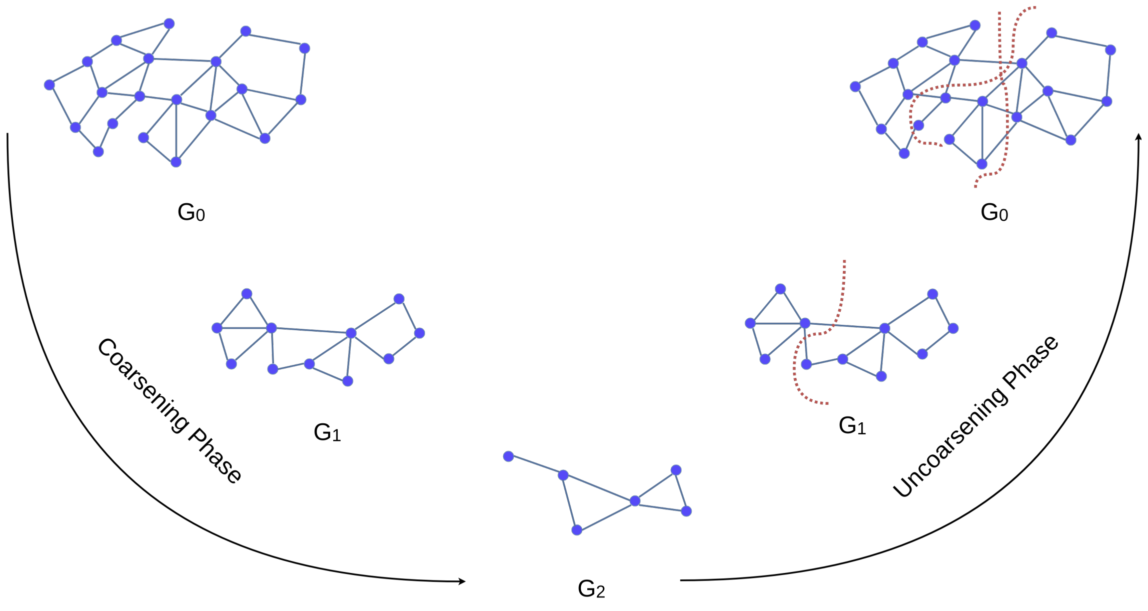

Figure 3.1: Multilevel Coarse Grid construction.

3.2.1 Dual Graph representation

It is possible to represent a grid as a Dual Graph $G = (V,E)$, where each vertex corresponds to grid element. Graphs can be weighted, for which each edge has an associated **weight**, typically given by a **weight function**. With E being the set of edges in a graph G, and V its set of vertices, one can store the weight $w(v_1,v_2)$ of the edge $(v_1,v_2) \in E \;\wedge\; v_1,v_2 \in V$. When graphs are unweighted, no weight is associated with an edge. The graph G can be a directed graph, where an edge (v_1,v_2) means that v_2 is adjacent to v_1 but v_1 is not adjacent to v_2. If G is an undirected graph, the edge (v_1,v_2) means that v_1 is adjacent to v_2, and vice-versa [4]. Our grid can be seen as an undirected and weighted graph, where an edge is connecting two vertices if a segment or face is shared, for a two- or three-dimensional grid respectively.

In order to have information to calculate the aspect-ratios of each control-volume, three values were defined: the *vertex-weight* (v^w), the *vertex-boundary-surface* (v^s) and the *vertex-volume* (v^v).

The *vertex-weight* represents the number of elements within the same *control-volume*. This number will increase with the successive number of executions of the coarsening/un-coarsening phase. The *vertex-boundary-surface* defines the number of segments or faces that are not shared by any other elements within the same *control-volume*. Therefore it is possible to know whether these elements belong to a boundary or not. The *vertex-volume* represents the area (in two-dimensional grids), or the volume (in three-dimensional grids) of a given *control-volume*. Lastly, one more metric was defined, the edge weight. This value corresponds to the length of the shared edge or the area of the shared face. The figure 3.2 contains a representation of a two-dimensional mesh and its dual graph.

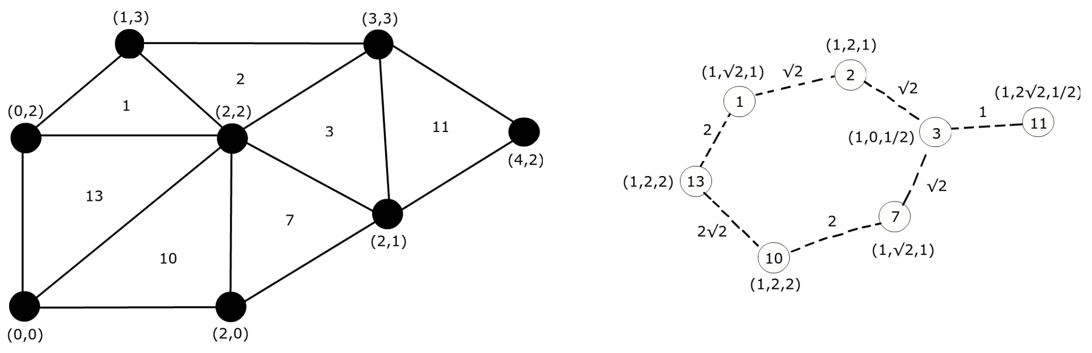

Figure 3.2: A two-dimensional mesh and the correspondent dual graph.

With this representation, one can approach such problem as a *k-way partitioning* of the vertices. Each partition must have a number of vertices that is between a range of values. These values are declared as *Lmin*, the lower limit, and *Lmax*, the upper limit. The process must be completed by following a particular *objective-function* [21, 27].

3.2.2 Coarsening Phase

As mentioned in section 3.2, the grid can be represented as a *Dual Graph* $G = (V, E)$. Each vertex V represents an element of the grid, and pairs of elements who share a segment or a face (for two- or three-dimensional grids) are connected by a given edge E. A sequence of smaller grids can be obtained $\{G_0, G_1, ..., G_n\}$. Each grid G_i is an approximation of the original grid G_0, and is calculated by merging pairs of vertices which are connected by a given edge. This method can be better visualized in figure 2.9. Therefore, having a grid G_i with a maximal independent set of edges I_i, it is possible to know that the next approximation G_{i+1} will have less $I_i - 1$ vertices. An independent set of edges of a graph is a set of edges no two of which are incident to the same vertex. An independent set is maximal if it is not possible to add any other edge to it without making two edges become incident on the same vertex [27].

After such operation, the properties v^w, v^s and v^v are updated: Having two vertices v_1, v_2, that are being collapsed to form a vertex u, one can declare the new properties, as,

- $u^w = v_1^w + v_2^w$
- $u^s = v_1^s + v_2^s$
- $u^v = v_1^v + v_2^v$

The connectivity information is also preserved. If both vertices, v_1 and v_2 are connected to the same vertex $v3$, (u_1, v_3) and (u_2, v_3), then a new edge is formed with a weight that can be represented as the sum of both edges weight $(u_1 + u_2, v_3)$. Notice that this won't happen if both vertices are not connected to the same vertex. In this case, only the weight of the connected edge is preserved.

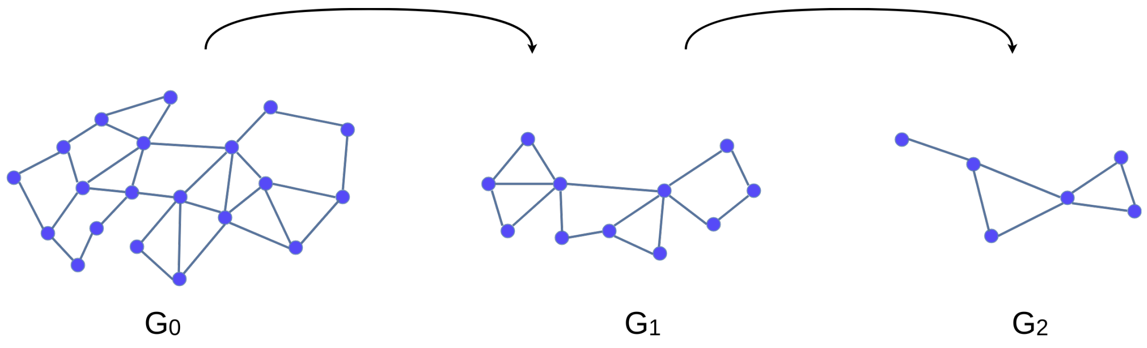

Figure 3.3: A visualisation of a possible coarsening phase.

One of the algorithms used to conduct this phase is called **Globular Matching**. It is based in the *Heavy-Edge Matching* algorithm described in section 2.9. This algorithm selects a maximal independent set of edges, trying to create a control volume from the pairs of vertices v_1, v_2 where $(v_1, v_2) \in E$, leads to the smallest aspect ratio. One can formulate the steps of the algorithm as follows,

Algorithm: Globular Matching

```
1 let $V$ be the set of unmatched vertices;
2 while $V \neq \emptyset$ do
3     let $e = (v, u)$ be the edge leading to the smallest aspect ratio ;
4     match $v$ and $u$;
5     add $e$ to the independent set;
6 end
```

The time complexity of this algorithm is $O(|E|)$, where $|E|$ represents the number of edges in a given G graph.

The second algorithm is called **Local Heaviest Approximation**, maintaining a greedy approach. This method arbitrarily selects a control volume, avoiding to keep them sorted by their weight. It follows the same idea of matching the randomly chosen vertex v_1 with a vertex v_2, that leads to the smallest aspect ratio. Hence, it maintains a time complexity of $O(|E|)$.

Algorithm: Local Heaviest Approximation

```
1 let $V$ be the set of unmatched vertices;
2 while $V \neq \emptyset$ do
3     let $v$ be an arbitrarily chosen vertex $\in V$ ;
4     let $e = (v, u)$ be the edge leading to the smallest aspect ratio ;
5     match $v$ and $u$;
6     add $e$ to the independent set;
7 end
```

The third coarsening method, Path Growing Algorithm, grows a set of disjoint paths. By randomly choosing an unmatched vertex, the edge that leads to the smallest aspect ratio is selected $e = (v,u)$, extending a path in that direction, with u being the next vertex from which the algorithm is going to grow a path of. A path is no longer extended when no more vertexes can be matched. In this case, the next random and unmatched vertex is selected, growing a path from there. While growing the paths, vertexes are alternatively added to two different matchings P_1 and P_2. The matching that leads to the best minimization of **F4** prevails [5, 25].

Notice that the time complexity of this algorithm is still $O(|E|)$, as every edge is analysed at most once.

Algorithm: Path Growing Algorithm

```
1  let $V$ be the set of unmatched vertices;
2  while $V \neq \emptyset$ do
3      let $v$ be an arbitrarily chosen vertex $\in V$ ;
4      let $alt$ be $true$;
5      let $P_1$ and $P_2$ be $\emptyset$;
6      while $\exists\ v,u \in V$ where $v,u$ can be matched do
7          let $e = (v,u)$ be the edge leading to the smallest aspect ratio ;
8          if $alt$ is $true$ then add $e$ to $P_1$ ;
9          else add $e$ to $P_2$;
10         $alt$ is $!alt$;
11         let $v$ be $u$;
12     end
13     match vertexes in $P$, where $P$ is min of $P_1, P_2$;
14 end
```

The fourth and last coarsening algorithm obtains a minimum matching in every coarsening level. Since the idea of our coarsening algorithms is to match pairs of vertexes in each level of approximation of our original grid G_i, the number of matching possibilities reduces significantly. As seen in section 2.2, by defining such premiss, it is possible to obtain a minimum matching in polynomial time. Therefore, by using a variance of the blossom algorithm [22], we can define our Minimum Approximation Algorithm as,

Algorithm: Minimum Approximation

```
1 let $E$ be the set of edges;
2 let $M'$ be the edges $\in E$ matched by the blossom algorithm;
3 while number of matched vertexes $< 0.25\times$ number of vertexes do
4     let $e = (v,u)$ be the edge in $M'$ leading to the smallest aspect ratio ;
5     match $v$ and $u$;
6     remove $e$ from $M'$;
7 end
```

Notice that only 25% of the total number of vertexes are matched. Such constraint increases the number of neighbours to be considered during the uncoarsening phase, increasing the probability of overall improvement as well. This algorithm has a time complexity of $O(|E||V|log|E|)$ [22].

3.2.3 Uncoarsening Phase

As mentioned before, the main purpose of the uncoarsening phase is to propagate the coarsest graph to the the original graph, going through the graphs $\{G_{n-1},...,G_1,G_0\}$, and refining the solution of each graph. The refinement process consists in moving vertices among control volumes, always respecting the constraints imposed by the *Lmin* and *Lmax* limits. Figure 3.4 shows possible adaptions. Such movements are an attempt in minimizing the objective function that is being followed throughout the multilevel construction.

Such action is possible because of three conditions. The use of a greedy algorithm to construct independent sets of edges may not lead to an optimal independent set. The objective function being followed may be different than the heuristic used to guide the multilevel construction. And finally, uncoarsed graphs have more levels of freedom than smaller coarsed graphs. The algorithm used to complete this stage can be described in the following steps,

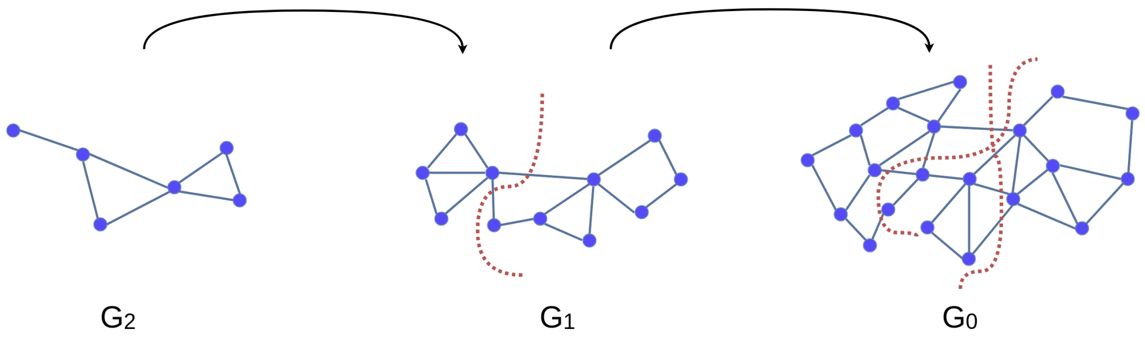

Figure 3.4: A visualisation of a possible uncoarsening phase.

```
let $V$ be the set of vertices;
while $\exists\, v \in V$ where $v$ can be moved and reduce a given objective function do
    for each random $v$ in $V$ do
        let $r =$ biggest reduction in moving $v$ into a different control volume $c$;
        if $r > 0$ then
            move $v$ to $c$;
        end
    end
end
```

This algorithm does not guarantee contiguous partitions, because vertices that are not adjacent to a control volume, can become part of it. In order to correct this problem, non-contiguous control volumes are split into different partitions. But this process may result in having control volumes that don't respect the *Lmin* constraint. In this case, after having a contiguous graph, small partitions are merged with adjacent partitions, respecting the established number of vertices limits within a control volume. The merging process of this phase is driven by the objective function to be optimized. In case of still having control volumes with fewer vertices than the established limit *Lmin*, some vertices are moved from large adjacent control volumes that can afford to lose those vertices.

3.3 Parallel Implementation

The multilevel graph partition algorithm described in section 3.2 can be implemented in parallel. Our algorithm distributes the mesh into p partitions, where p represents the total number of processors. This process distributes elements among processors, resulting in a balanced number of vertices between partitions. Each processor operates in its local partition, without communicating with any other processor during this process. The serial algorithm for multilevel coarse grid construction is then applied in each one of these subdomains. This approach creates good aspect ratios in the interior of the subdomains. But because boundary elements are not capable of fusing with elements in different subdomains, those areas may suffer from low quality control volumes.

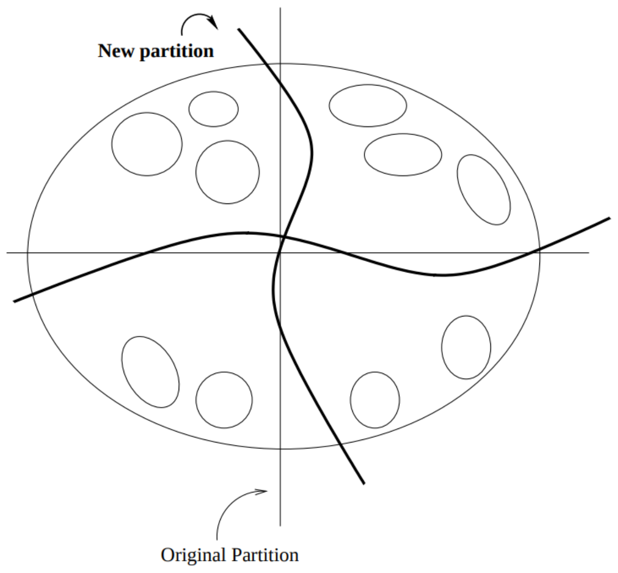

Figure 3.5: Parallel partition [27].

One solution to this problem is to allow elements in the boundary areas to participate in refinement iterations with elements from different partitions. This approach may limit the efficiency of the parallel algorithm, because it includes a overhead of communication and synchronization.

Our solution to this problem is to use an adaptive graph partitioning algorithm [27]. In this approach, the elements near to a boundary move closer to the interior of the partition. This method will cause fused elements in other subdomains to move to the same subdomain as well. This method can be visualized in figure 3.5. The local refinement, similar to what is done in the serial refinement process, is performed until the overall quality of the coarse grid does not improve any further.

3.4 Execution Conditions

The performance of the previously described methods was evaluated in an HPC cluster named Delta. This cluster is installed locally, at Cranfield University. There are 11 6U chassis, and 12 nodes housed per each one of them. These chassis are spread over five racks, what makes a total of 118 compute nodes with two Intel E5-2620 v4 (Broadwell) CPUs each. Such CPU is built with 16 cores, and 128 GB of shared memory. Hence, this machine has a total of 1888 available cores. Delta compute nodes are connected via an Infiniband EDR low-latency interconnect [32].

The grids available to execute and experiment the algorithm can be consulted on table 3.1. These are 3D tetrahedral meshes, with each one being a representation of an air-plane wing, and their number of elements varies considerably.

Table 3.1: Number of elements per mesh.

Name	# Elements
M6	94,493
F22	428,748
F16	1,124,648

3.5 Validation

The starting point of this thesis was based on work developed by I. Moulitsas and G. Karypis in 2001 [27]. In this section, the a software implementation for generating a sequence of coarse grids, MGridGen / ParMGridGen, are going to be validated. Notice that the multilevel construction methodology for generating coarser grids was described in the previous sections. The values of the objective functions F_2 and F_3 are going to be analysed. We will obtain results regarding a serial and several parallel executions with **2**, **4**, **8**, **16**, **32** and **64** processors. Table 3.2 contains the information regarding this verification.

Table 3.2: Quality measures on Delta on 1, 2, 4, 8, 16, 32 and 64 processors.

	M6		F22		F16	
# Processes	F_3	F_2	F_3	F_2	F_3	F_2
1	2.25e+01	1.83e+06	2.64e+01	8.33e+06	2.29e+01	2.04e+07
2	2.26e+01	1.83e+06	3.17e+01	8.30e+06	2.84e+01	2.04e+07
4	2.35e+01	1.82e+06	2.33e+01	8.30e+06	2.84e+01	2.04e+07
8	2.25e+01	1.82e+06	2.52e+01	8.28e+06	2.84e+01	2.03e+07
16	2.26e+01	1.81e+06	3.08e+01	8.26e+06	2.50e+01	2.03e+07
32	2.26e+01	1.81e+06	3.17e+01	8.23e+06	2.90e+01	2.02e+07
64	2.26e+01	1.80e+06	3.49e+01	8.22e+06	2.24e+01	2.02e+07

Table 3.3: Quality measures on Cray T3E on 1, 2, 4, 8, 16, 32 and 64 processors.

	M6		F22		F16	
# Processes	F_3	F_2	F_3	F_2	F_3	F_2
1	2.43e+01	1.83e+06	—	—	—	—
2	2.26e+01	1.83e+06	—	—	—	—
4	2.25e+01	1.82e+06	2.71e+01	8.29e+06	—	—
8	2.27e+01	1.82e+06	2.93e+01	8.28e+06	2.24e+01	2.02e+07
16	2.26e+01	1.81e+06	2.31e+01	8.25e+06	2.64e+01	2.02e+07
32	2.26e+01	1.80e+06	2.36e+01	8.23e+06	7.11e+01	2.01e+07
64	2.26e+01	1.80e+06	2.40e+01	8.21e+06	2.28e+01	2.01e+07

By looking at the tables 3.2, 3.3 and 3.4, it is possible to state that the quality measurements present similar results in the F_2 values. Even though that the results of F_3 are somewhat different from the original results, this is due to the fact that this value is much more influenced by the random generation approach of each machine. Keeping that in mind, it is safe to consider that the obtained results are valid.

Table 3.4: Quality measures on BEO on 1, 2, 4, 8, and 16 processors.

	M6		F22		F16	
# Processes	F_3	F_2	F_3	F_2	F_3	F_2
1	2.70e+01	1.82e+06	2.55e+01	8.33e+06	2.28e+01	2.04e+07
2	2.29e+01	1.82e+06	2.32e+01	8.30e+06	2.84e+01	2.03e+07
4	3.19e+01	1.82e+06	2.55e+01	8.29e+06	2.84e+01	2.03e+07
8	2.27e+01	1.81e+06	2.41e+01	8.28e+06	2.28e+01	2.03e+07
16	2.26e+01	1.81e+06	2.31e+01	8.25e+06	2.84e+01	2.02e+07

3.6 Scalability

The beginning of the MGridGen / ParMGridGen development was roughly 17 years ago. Hence, it was expected to have runs, under the same conditions, with a smaller execution time. Since the computational power and the amount of available memory are strongly connected with Moore's Law, previously explained in section 2.3, we can presume that after this time, the computational power of a recent machine increased considerably. Thereby, in this section, we're going to study the contrast between execution times obtained by I. Moulitsas and G. Karypis in 2001 [27] and today. Their results were obtained in two different machines. One was a CRAY T3E-1200 with 1024 EV6 Alpha processors and 512MB of memory at each processor. This machine was running at 600MHz. The second machine had 16 processors with workstations connected through a 100MBit Ethernet switch. It had Intel Pentium III processors running at 650 MHz with 1GB of memory. This architecture was referred as "BEO".

The execution time of each mesh on Delta, CRAY T3E and BEO can be found on table 3.5. When looking at Delta results, it can be seen that the computational time of M6 and F22 partitioning increases with 32 and 64 processes. By following *Amdahl's law*, since the number of processors is increasing and the problem size is fixed, the execution time would increase until there's no space of enhancement. But as mentioned before, this law is given as optimistic, and doesn't take several variables into account. The time increase is related with the high communication cost among a larger number of processes. Following

Table 3.5: Execution times in seconds with 1, 2, 4, 8, 16, 32 and 64 processors on Delta, CRAY T3E and BEO clusters.

	Delta			CRAY T3E			BEO	
# Processes	M6	F22	F16	M6	F22	F16	M6	F22
1	1.69	10.23	30.91	80.66	—	—	32.74	173.61
2	2.35	14.53	44.34	103.17	—	—	46.14	246.66
4	1.15	6.47	19.98	50.43	256.13	—	28.21	153.21
8	0.64	3.55	10.56	22.30	125.21	—	14.18	74.71
16	0.37	2.03	5.97	9.95	61.06	163.14	18.64	50.55
32	1.02	2.18	3.51	4.38	29.71	90.08	—	—
64	1.35	2.45	2.89	2.24	14.86	40.47	—	—

this idea, and keeping in mind that each Delta CPU has 16 cores, execution times with more than 16 processes will certainly contain communication cost. But because the F16 mesh has more edges and vertices, the execution time of that mesh continues on scaling. The last statement is supported by *Gustafson's law*, described in section 2.5. It is also possible to see that Delta produced considerably lower computational times when running the program with all three mesh. Each one of the grids could fit in memory, overcoming the memory limitations of *CRAY T3E*.

Chapter 4

Results

The development of new coarsening approaches requires analysis, they need to be studied and compared. The effect of new objective functions is going to be analysed as well. The execution conditions to test such changes are the same as the ones presented in section 3.4. These results were obtained with 1, 2, 4, 8, 16, 32 and 64 processes.

As mentioned in section 3.1, the overall grid quality can be measured by the several objective functions. The sum, weighted sum, and squared sum of Aspect Ratios are going to be presented in order to perform an analysis of the overall grid quality. Nevertheless, the coarsening factor of a coarse grid is going to be presented as well. Less coarsened grids present less elements, and therefore tend to present lower objective function values. The coarsening factor can be obtained by dividing the number of elements of the original grid by the number of elements of the coarsened grid, telling us how many times the grid was coarsened. The *Lmin* and *Lmax* limits to obtain these results were established as **one** and **four** respectively.

4.1 F3 + F4

As mentioned in section 3.6, the refinement driven by the $F_3 + F_2$ combination was already studied. Hence, a new combination $F_3 + F_4$ will be studied and analysed. The traditional algorithm *Globular Matching* was used to drive the coarsening phase during this study.

In this section, for the sake of simplicity, figures and tables are going to label the $F_3 + F_2$ and $F_3 + F_4$ combinations as *Traditional* (trad) and *Squared Sum* (ssum) respectively. The reader should keep in mind that these are, in fact, combinations of two objective functions.

4.1.1 Serial Algorithm Evaluation

The following results were obtained by the serial multilevel coarse grid construction algorithm. These results are going to evaluate the overall grid quality and the algorithm performance.

Table 4.1: Comparison between the *Traditional* and *Squared Sum* combinations.

	M6		F22		F16	
	trad	ssum	trad	ssum	trad	ssum
Execution Time (s)	1.69	1.54	10.23	10.05	30.91	29.96
Coarsening Factor	3.47	3.15	3.47	3.45	3.46	3.47
F_1	5.25e+05	5.67e+05	2.38e+06	2.40e+06	5.87e+06	5.86e+06
F_2	1.82e+06	1.81e+06	8.33e+06	8.33e+06	2.04e+07	2.04e+07
F_3	2.25e+01	2.38e+01	2.63e+01	2.33e+01	2.28e+01	2.55e+01
F_4	6.98e+06	5.52e+06	1.70e+07	1.56e+07	8.53e+05	7.64e+05

Regarding to execution time, table 4.1 shows that the *Squared Sum* combination is slightly faster. But it never gets more than 10% faster. Despite the difference of coarsening factors in the *M6* grid, both combinations obtained similar coarsening factors with *F22* and *F16* meshes. When it comes to overall grid quality, looking at *M6* results, even though that *Squared Sum* produced a larger grid, it can be seen that the F_1, F_2 and F_4 values are lower. This is not true for F_3. Both F_1 and F_2 metrics present similar results in the other two grids. The F_4 function produces better results in the *Squared Sum* combination for all three grids. Following the same idea, it would be expected that the *Traditional* method would produce better results when looking at the F_2 function, but this is not true. It is worth to mention that the F_3 results were not consistent. The *Traditional* combination produced better values for two meshes (*M6* and *F16*), and the *Squared Sum* combination obtained better results in *F22*.

4.1.2 Parallel Algorithm Evaluation

The following results were obtained by the parallel multilevel coarse grid construction algorithm. Identically to the previous subsection, these results are going to evaluate the overall grid quality and the algorithm performance. The overall quality of *M6* will be analysed first, followed by *F22* and *F16*.

Table 4.2: *M6* quality measurements comparison between the *Traditional* and *Squared Sum* combinations on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.25e+01	6.98e+06	1.83e+06	2.38e+01	5.52e+06	1.81e+06
2	2.26e+01	6.94e+06	1.82e+06	2.38e+01	5.10e+06	1.79e+06
4	2.35e+01	6.93e+06	1.82e+06	2.38e+01	5.14e+06	1.78e+06
8	2.34e+01	6.93e+06	1.82e+06	2.38e+01	5.14e+06	1.78e+06
16	2.26e+01	6.85e+06	1.81e+06	2.25e+01	5.62e+06	1.77e+06
32	2.26e+01	6.78e+06	1.80e+06	2.24e+01	5.71e+06	1.77e+06
64	2.26e+01	6.81e+06	1.80e+06	2.24e+01	5.77e+06	1.77e+06

Table 4.2 shows a comparison of the overall quality of grids obtained from **M6**. This is a comparison between the *Traditional* and *Squared Sum* combinations. *Squared Sum* was shown to obtain better results in F_4 and F_2 for all runs. When looking at F_3 results, we can see some variance, but the difference is never larger than **1.3**. It is worth to emphasize, that similarly to the results obtained with the serial algorithm, the *Squared Sum* combination was capable of minimizing the **F2** objective function better. When looking at table 4.5 it can be seen that for M6 this combination is producing larger grids than the *Traditional* combination. Despite of having larger grids, the sums of aspect ratios, F2 and F4, are presenting lower results.

Table 4.3: *F22* quality measurements comparison between the *Traditional* and *Squared Sum* combinations on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.63e+01	1.70e+07	8.33e+06	2.33e+01	1.56e+07	8.33e+06
2	3.16e+01	1.70e+07	8.30e+06	2.33e+01	1.45e+07	8.28e+06
4	2.32e+01	1.69e+07	8.29e+06	2.41e+01	1.46e+07	8.27e+06
8	2.52e+01	1.70e+07	8.28e+06	2.41e+01	1.45e+07	8.26e+06
16	3.08e+01	1.69e+07	8.25e+06	2.44e+01	1.46e+07	8.24e+06
32	3.16e+01	1.69e+07	8.23e+06	2.71e+01	1.48e+07	8.22e+06
64	3.48e+01	1.69e+07	8.21e+06	3.24e+01	1.53e+07	8.20e+06

The data in table 4.3 shows again a comparison of the overall grid quality between the *Traditional* and *Squared Sum* combinations, but this time from grids obtained from *F22*. It can be seen that the *Squared Sum* combination produces better results in F_2 and F_4. For this combination only one value of F_3 is worst, in a grid obtained from a run with 16 processes.

Table 4.4: *F16* quality measurements comparison between the *Traditional* and *Squared Sum* combinations on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.28e+01	8.53e+05	2.04e+07	2.55e+01	7.64e+05	2.04e+07
2	2.84e+01	8.45e+05	2.03e+07	2.84e+01	7.03e+05	2.03e+07
4	2.84e+01	8.44e+05	2.03e+07	2.27e+01	7.03e+05	2.03e+07
8	2.84e+01	8.44e+05	2.03e+07	2.24e+01	7.22e+05	2.03e+07
16	2.49e+01	8.40e+05	2.03e+07	2.84e+01	7.25e+05	2.03e+07
32	2.90e+01	8.40e+05	2.02e+07	2.44e+01	7.39e+05	2.02e+07
64	2.24e+01	8.29e+05	2.01e+07	2.89e+01	7.32e+05	2.02e+07

Finally, table 4.3 shows the same comparison from grids obtained from *F16*. It can be seen that F_2 values are quite similar in both combinations, but F_4 presents smaller values on *Squared Sum*. F_3 presented some fluctuations, ranging from $2.28e+01$ to $2.89e+01$, and *Squared Sum* presents better results on runs with 4, 8 and 32 processes.

As seen in table 4.5, and focusing on runs following the *Squared Sum* combination, the coarsening factor of grids obtained from the M6 mesh is smaller, but it becomes identical in the remaining meshes. The execution time of runs following this combination is always smaller.

Table 4.5: Execution times in seconds and Coarsening Factors comparison between the *traditional* and *Squared Sum* combinations on 1, 2, 4, 8, 16, 32 and 64 processors.

	Execution time (s)						Coarsening Factor					
	Traditional			Squared Sum			Traditional			Squared Sum		
# Processes	M6	F22	F16	M6	F22	F16	M6	F22	F16	M6	F22	F16
1	1.69	10.24	30.81	1.54	10.11	29.79	3.47	3.47	3.46	3.13	3.44	3.47
2	2.33	14.70	44.39	2.17	13.94	42.86	3.45	3.42	3.43	2.86	3.34	3.42
4	1.15	6.50	19.76	1.11	6.33	18.76	3.43	3.40	3.41	2.85	3.32	3.41
8	0.61	3.48	10.47	0.61	3.34	9.78	3.38	3.36	3.39	2.83	3.30	3.43
16	0.36	2.06	5.95	0.36	1.92	5.53	3.31	3.30	3.35	2.81	3.24	3.35
32	0.26	1.07	2.81	0.22	0.94	2.68	3.25	3.25	3.27	2.78	3.20	3.28
64	0.19	0.74	1.44	0.15	0.65	1.33	3.19	3.16	3.24	2.77	3.24	3.24

4.2 Local Heaviest Approximation

In this section, the results obtained by the Maximum Local Matching algorithm are going to be analysed. Since this algorithm is meant to drive the coarsening phase, the results were obtained by following the traditional combination, $F_3 + F_2$, of objective functions.

Starting by analysing the overall quality of grids obtained from M6 in table 4.6, it becomes evident that this matching algorithm produced worst results in all of F_2 values. The same happened in most of F_4 values, being able to produce better results in runs with 8, 16 and 32 processes. In regards to F_3 metric, better values were obtained in three runs again, with 16, 32 and 64 processes. Both F_3 and F_4 were not very consistent, since they obtained three cases with better results with Local Heaviest Approximation and four cases

Table 4.6: *M6* quality measurements comparison between Globular Matching and Local Heaviest Approximation on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Local Heaviest Approximation		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.25e+01	6.98e+06	1.83e+06	2.33e+01	7.13e+06	1.84e+06
2	2.26e+01	6.94e+06	1.82e+06	2.29e+01	8.84e+06	1.45e+06
4	2.35e+01	6.93e+06	1.82e+06	2.55e+01	6.97e+06	1.83e+06
8	2.34e+01	6.93e+06	1.82e+06	2.55e+01	6.92e+06	1.83e+06
16	2.26e+01	6.85e+06	1.81e+06	2.24e+01	6.83e+06	1.82e+06
32	2.26e+01	6.78e+06	1.80e+06	2.24e+01	6.76e+06	1.81e+06
64	2.26e+01	6.81e+06	1.80e+06	2.24e+01	6.82e+06	1.80e+06

with better results with Globular Matching.

Table 4.7 shows several data regarding to the overall quality of grids derived from F22. Focusing on F_4 and F_2, the Local Heaviest Approximation algorithm, obtained worst results in every execution. F_3 was not consistent again, showing some fluctuations and obtaining better results in 3 of 7 runs.

Table 4.7: *F22* quality measurements comparison between Globular Matching and Local Heaviest Approximation on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Local Heaviest Approximation		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.63e+01	1.70e+07	8.33e+06	2.36e+01	1.74e+07	8.41e+06
2	3.16e+01	1.70e+07	8.30e+06	2.39e+01	1.72e+07	8.37e+06
4	2.32e+01	1.69e+07	8.29e+06	2.86e+01	1.73e+07	8.35e+06
8	2.52e+01	1.70e+07	8.28e+06	2.90e+01	1.72e+07	8.33e+06
16	3.08e+01	1.69e+07	8.25e+06	2.32e+01	1.72e+07	8.29e+06
32	3.16e+01	1.69e+07	8.23e+06	2.31e+01	1.71e+07	8.26e+06
64	3.48e+01	1.69e+07	8.21e+06	2.58e+01	1.71e+07	8.23e+06

Thirdly, table 4.8 shows the same information about the F16 mesh. Similarly to what happened in the previous grid, worst results were obtained in every run, when looking at F_4 and F_2. Again, variances were encounter in regards to F_3.

Table 4.8: *F16* quality measures comparison between Globular Matching and Local Heaviest Approximation on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Local Heaviest Approximation		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.28e+01	8.53e+05	2.04e+07	2.27e+01	8.61e+05	2.06e+07
2	2.84e+01	8.45e+05	2.03e+07	2.84e+01	8.58e+05	2.05e+07
4	2.84e+01	8.44e+05	2.03e+07	2.24e+01	8.59e+05	2.05e+07
8	2.84e+01	8.44e+05	2.03e+07	2.45e+01	8.54e+05	2.04e+07
16	2.49e+01	8.40e+05	2.03e+07	2.50e+01	8.49e+05	2.04e+07
32	2.90e+01	8.40e+05	2.02e+07	2.23e+01	8.44e+05	2.03e+07
64	2.24e+01	8.29e+05	2.01e+07	2.32e+01	8.35e+05	2.02e+07

Finally, table 4.9 shows the execution times and Coarsening Factors of both Globular Matching and Local Heaviest Approximation. When observing this table, it can be seen that the fact of being avoiding to sort edges by their weight, produces faster runs in every case. The difference between Coarsening Factors was shown to be more evident in runs with less processes.

Table 4.9: Execution times in seconds and Coarsening Factors comparison between Globular Matching and Local Heaviest Approximation on 1, 2, 4, 8, 16, 32 and 64 processors.

	Execution time (s)						Coarsening Factor					
	Globular Matching			Max Local Matching			Globular Matching			Max Local Matching		
# Processes	M6	F22	F16	M6	F22	F16	M6	F22	F16	M6	F22	F16
1	1.69	10.24	30.81	1.41	8.98	27.04	3.47	3.47	3.46	3.59	3.60	3.60
2	2.33	14.70	44.39	2.07	13.32	41.05	3.45	3.42	3.43	3.56	3.53	3.56
4	1.15	6.50	19.76	1.03	5.98	18.01	3.43	3.40	3.41	3.53	3.50	3.54
8	0.61	3.48	10.47	0.55	3.22	9.88	3.38	3.36	3.39	3.47	3.45	3.50
16	0.36	2.06	5.95	0.33	1.94	5.62	3.31	3.30	3.35	3.37	3.36	3.43
32	0.26	1.07	2.81	0.23	0.94	2.64	3.25	3.25	3.27	3.30	3.30	3.33
64	0.19	0.74	1.44	0.13	0.70	1.49	3.19	3.16	3.24	3.22	3.24	3.28

4.3 Path Growing Algorithm

Similarly to the previous section, the results obtained by the *Path Growing Algorithm* are going to be analysed. The first set of results was collected from executions driven by $F_3 + F_4$. In order to understand how the *Path Growing Algorithm* and the $F_3 + F_2$ combination work together, a second set of results, acquired from executions driven by this combination, will be studied as well.

4.3.1 Traditional

Table 4.10: *M6* quality measurements comparison between Globular Matching and Path Growing Algorithm on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Path Growing Algorithm		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.25e+01	6.98e+06	1.83e+06	2.24e+01	6.63e+06	1.78e+06
2	2.26e+01	6.94e+06	1.82e+06	2.26e+01	6.55e+06	1.77e+06
4	2.35e+01	6.93e+06	1.82e+06	2.24e+01	6.44e+06	1.77e+06
8	2.34e+01	6.93e+06	1.82e+06	2.25e+01	6.38e+06	1.75e+06
16	2.26e+01	6.85e+06	1.81e+06	2.24e+01	6.26e+06	1.74e+06
32	2.26e+01	6.78e+06	1.80e+06	2.24e+01	6.25e+06	1.73e+06
64	2.26e+01	6.81e+06	1.80e+06	2.26e+01	6.20e+06	1.72e+06

Table 4.10 shows a comparison between quality measurements of results acquired by the *Globular Matching* and *Path Growing Algorithm*. It can be seen that the last matching algorithm obtained better results for this grid in very metric. F_4 is showing 5% to 10% smaller values, whereas the F_2 objective function obtained an improvement of 2% to 5%. The F_3 metric presented similar values in two runs, with 2 and 64 processes. Notice that in the Path Growing Algorithm, this objective function presented values that were very stable whilst the number of processes was increasing.

When looking at table 4.10, which shows the same comparison with a different grid, *F22*, it can be seen that better results were still obtained on F_4 and F_2. However, $F3$ presents an uncommon difference in the execution with 32 processes. The difference in

Table 4.11: *F22* quality measurements comparison between Globular Matching and Path Growing Algorithm on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Path Growing Algorithm		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.63e+01	1.70e+07	8.33e+06	2.31e+01	1.64e+07	8.14e+06
2	3.16e+01	1.70e+07	8.30e+06	2.32e+01	1.63e+07	8.09e+06
4	2.32e+01	1.69e+07	8.29e+06	2.74e+01	1.62e+07	8.05e+06
8	2.52e+01	1.70e+07	8.28e+06	2.71e+01	1.62e+07	8.01e+06
16	3.08e+01	1.69e+07	8.25e+06	2.90e+01	1.61e+07	7.95e+06
32	3.16e+01	1.69e+07	8.23e+06	4.48e+01	1.60e+07	7.90e+06
64	3.48e+01	1.69e+07	8.21e+06	2.68e+01	1.58e+07	7.86e+06

the F_4 function becomes less evident, with improvements ranging from 4% to 7% only, and the F_2 function presenting roughly the same percentage of difference. The F_3 function, however, was not as stable as it was in the previous grid, presenting 3 larger values out of 7.

The results of the third grid, *F16*, can be consulted in table 4.12, which presents similar results to what was seen in the previous grid. Again, we can see smaller values in the F_4 and F_2 objective functions, and 2 larger values on F_3.

Table 4.12: *F16* quality measurements comparison between Globular Matching and Path Growing Algorithm on 1, 2, 4, 8, 16, 32 and 64 processors.

	Globular Matching			Path Growing Algorithm		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.28e+01	8.53e+05	2.04e+07	2.24e+01	8.05e+05	2.00e+07
2	2.84e+01	8.45e+05	2.03e+07	2.27e+01	8.02e+05	1.99e+07
4	2.84e+01	8.44e+05	2.03e+07	2.24e+01	8.04e+05	1.98e+07
8	2.84e+01	8.44e+05	2.03e+07	2.39e+01	8.01e+05	1.97e+07
16	2.49e+01	8.40e+05	2.03e+07	2.64e+01	7.92e+05	1.96e+07
32	2.90e+01	8.40e+05	2.02e+07	2.64e+01	7.79e+05	1.94e+07
64	2.24e+01	8.29e+05	2.01e+07	2.64e+01	7.70e+05	1.93e+07

Table 4.13 shows a comparison between Execution Times and Coarsening Factors.

This algorithm presents a smaller execution time in the serial run, and larger execution times in every parallel run. The algorithm produced smaller grids in the overall, increasing the difference in executions with a larger number of processes.

Table 4.13: Execution times in seconds and Coarsening Factors comparison between Globular Matching and Local Heaviest Approximation on 1, 2, 4, 8, 16, 32 and 64 processors.

	Execution time (s)						Coarsening Factor					
	Globular Matching			Path Growing Alg.			Globular Matching			Path Growing Alg.		
# Processes	M6	F22	F16	M6	F22	F16	M6	F22	F16	M6	F22	F16
1	1.69	10.24	30.81	1.43	10.14	30.00	3.47	3.47	3.46	3.05	3.04	3.06
2	2.33	14.70	44.39	2.72	16.03	49.91	3.45	3.42	3.43	2.97	2.95	3.01
4	1.15	6.50	19.76	1.30	7.38	21.90	3.43	3.40	3.41	2.88	2.88	2.95
8	0.61	3.48	10.47	0.70	3.95	11.66	3.38	3.36	3.39	2.78	2.78	2.86
16	0.36	2.06	5.95	0.44	2.26	6.67	3.31	3.30	3.35	2.65	2.66	2.75
32	0.26	1.07	2.81	0.23	1.09	3.11	3.25	3.25	3.27	2.54	2.57	2.62
64	0.19	0.74	1.44	0.12	0.59	1.66	3.19	3.16	3.24	2.45	2.49	2.55

4.3.2 Squared Ratio

Table 4.14: *M6* quality measurements comparison between Traditional and Squared Sum on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.24e+01	6.63e+06	1.78e+06	2.28e+01	5.58e+06	1.77e+06
2	2.26e+01	6.55e+06	1.77e+06	2.26e+01	5.15e+06	1.75e+06
4	2.24e+01	6.44e+06	1.77e+06	2.24e+01	5.17e+06	1.74e+06
8	2.25e+01	6.38e+06	1.75e+06	2.24e+01	5.17e+06	1.73e+06
16	2.24e+01	6.26e+06	1.74e+06	2.24e+01	5.19e+06	1.71e+06
32	2.24e+01	6.25e+06	1.73e+06	2.24e+01	5.35e+06	1.71e+06
64	2.26e+01	6.20e+06	1.72e+06	2.24e+01	5.44e+06	1.69e+06

The results of *M6* executions, with the Path Growing Algorithm as the matching method, and Squared Sum as the objective functions to be followed, can be viewed in table 4.14. These results can be represented as a comparison of the new algorithm working with the *Traditional* and *Squared Sum* combinations. When looking at this table, it can be seen that the *Squared Sum* tends to produce better F_3 results. The table also shows that every F_4 and F_2 values are lower with this combination. Notice that a better minimization of the F_2 value was obtained, similarly to what happened previously, with a different matching algorithm.

Table 4.15: *F22* quality measurements comparison between Traditional and Squared Sum on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.31e+01	1.64e+07	8.14e+06	2.28e+01	1.54e+07	8.13e+06
2	2.32e+01	1.63e+07	8.09e+06	2.29e+01	1.43e+07	8.07e+06
4	2.74e+01	1.62e+07	8.05e+06	2.80e+01	1.44e+07	8.03e+06
8	2.71e+01	1.62e+07	8.01e+06	4.44e+01	1.46e+07	7.98e+06
16	2.90e+01	1.61e+07	7.95e+06	2.39e+01	1.44e+07	7.93e+06
32	4.48e+01	1.60e+07	7.90e+06	3.16e+01	1.43e+07	7.89e+06
64	2.68e+01	1.58e+07	7.86e+06	2.47e+01	1.47e+07	7.85e+06

Table 4.16: *F16* quality measurements comparison between Traditional and Squared Sum on 1, 2, 4, 8, 16, 32 and 64 processors.

	Traditional			Squared Sum		
# Processes	F_3	F_4	F_2	F_3	F_4	F_2
1	2.24e+01	8.05e+05	2.00e+07	2.24e+01	7.55e+05	2.00e+07
2	2.27e+01	8.02e+05	1.99e+07	2.64e+01	6.99e+05	1.99e+07
4	2.24e+01	8.04e+05	1.98e+07	2.64e+01	7.07e+05	1.98e+07
8	2.39e+01	8.01e+05	1.97e+07	2.28e+01	7.04e+05	1.97e+07
16	2.64e+01	7.92e+05	1.96e+07	2.64e+01	7.10e+05	1.96e+07
32	2.64e+01	7.79e+05	1.94e+07	2.46e+01	7.02e+05	1.94e+07
64	2.64e+01	7.70e+05	1.93e+07	2.64e+01	7.15e+05	1.93e+07

Table 4.15 shows the same comparison on a different grid, *F22*. F_4 and F_2 show

improvements from the *Traditional* combination of objective functions again. We can see a larger improvement on F_4, but smaller values of F_2 in every run.

Thirdly, results of the *F16* grid were gathered in table 4.16. F_2 shown similar values in both combinations. F_4, identically to what happened before, obtained lower values in every run. F_3 presented only two larger values.

The Execution times and Coarsening Factors of all runs can be seen in table 4.17. Faster runs were obtained whilst using the Squared Sum combination of objective functions. The Coarsening Factor was also smaller, what means that this combination produced larger and less coarsened grids.

Table 4.17: Execution times in seconds and Coarsening Factors comparison between Traditional and Squared Sum on 1, 2, 4, 8, 16, 32 and 64 processors.

	Execution time (s)						Coarsening Factor					
	Traditional			Squared Sum			Traditional			Squared Sum		
# Processes	M6	F22	F16	M6	F22	F16	M6	F22	F16	M6	F22	F16
1	1.62	10.14	30.00	1.43	9.62	29.93	3.05	3.04	3.06	2.87	3.01	3.05
2	2.72	16.03	49.91	2.48	15.19	47.11	2.97	2.95	3.01	2.54	2.89	2.98
4	1.30	7.38	21.90	1.18	6.89	20.86	2.88	2.88	2.95	2.47	2.80	2.92
8	0.70	3.95	11.66	0.66	3.67	11.04	2.78	2.78	2.86	2.40	2.71	2.83
16	0.44	2.26	6.67	0.39	2.12	6.18	2.65	2.66	2.75	2.33	2.61	2.73
32	0.23	1.09	3.11	0.22	1.08	2.89	2.54	2.57	2.62	2.27	2.52	2.60
64	0.12	0.59	1.66	0.10	0.55	1.59	2.45	2.49	2.55	2.21	2.44	2.53

4.4 Minimum Approximation Matching

In this section, results of the serial *Minimum Approximation Matching* algorithm are going to be analysed. The first results are a comparison between the *Globular Matching* algorithm and the *Minimum Approximation Matching* algorithm, with both following the traditional $F_3 + F_2$ objective functions. In a second instance, the effect of the new combination $F_3 + F_4$ is going to be studied as well.

Table 4.18: Comparison between the *Globular Matching* and *Minimum Approximation Matching*, following $F_3 + F_2$.

	M6		F22		F16	
	G. Match.	Min Ap.	G. Match.	Min Ap.	G. Match.	Min Ap.
Execution Time (s)	1.69	4.90	10.23	94.44	30.91	182.97
Coarsening Factor	3.47	3.14	3.47	3.13	3.46	3.12
F_1	5.25e+05	5.69e+05	2.38e+06	2.59e+06	5.87e+06	6.37e+06
F_2	1.82e+06	1.79e+06	8.33e+06	8.17e+06	2.04e+07	2.00e+07
F_3	2.25e+01	2.24e+01	2.63e+01	2.32e+01	2.28e+01	2.64e+01
F_4	6.98e+06	6.75e+06	1.70e+07	1.65e+07	8.53e+05	8.26e+05

In table 4.18, it is evident that the execution time of our mesh partitioning algorithm increases with the new *Minimum Approximation Matching* method. We can see an increase on the execution time of around 4, 9 and 5 times in *M6*, *F22* and *F16* respectively. This algorithm tends to produce less coarsened grids as well. Nevertheless, when looking at the overall grid quality, the values of F_2 and F_4 are always lower, with F_3 only obtaining a larger value in the *F16* mesh. F_1 obtained larger values in each one of the three grids. This can be explained with the fact of having larger grids, and therefore, having more control volumes. Notice that despite such condition, F_2 and F_4 still obtained better results.

Table 4.19: Comparison between the *Minimum Approximation Matching* algorithm, following $F_3 + F_2$ and $F_3 + F_4$.

	M6		F22		F16	
	trad	ssum	trad	ssum	trad	ssum
Execution Time (s)	4.90	4.59	94.44	99.50	182.97	181.30
Coarsening Factor	3.14	2.89	3.13	3.11	3.12	3.13
F_1	5.69e+05	6.04e+05	2.59e+06	2.61e+06	6.37e+06	6.37e+06
F_2	1.79e+06	1.78e+06	8.17e+06	8.17e+06	2.00e+07	2.00e+07
F_3	2.24e+01	2.24e+01	2.32e+01	2.32e+01	2.64e+01	2.64e+01
F_4	6.75e+06	5.47e+06	1.65e+07	1.52e+07	8.26e+05	7.53e+05

In table 4.19 we can see the effect of *Minimum Approximation Matching* algorithm following F_3 + F_4, comparing it with the previous results. Despite of having slightly faster executions with *M6* and *F16*, the execution of the same algorithm on *F22* took approximately 5 more seconds. The coarsening factors were also similar on *F22* and *F16*, but smaller on *M6*. F_2 produced similar values with *F22* and *F16*, but a smaller value with *M6*. F_3 produced the same values with every grid, but we can see improvements in the F_4 metric. The results obtained in F_1 can, again, be explained by the same factor.

Chapter 5

Conclusions

Starting by the $F_3 + F_4$ combination of objective functions, and comparing it to the traditional combination, $F_3 + F_2$, we can see that it produces larger grids, but in a slightly smaller execution time. Despite of producing grids with more elements, $F_3 + F_4$ is obtaining better results in most of the overall grid quality metrics. Even when comparing the F_2 results obtained by both combinations, we can see lower values with the new combination. Notice that $F_3 + F_4$ is not directly minimizing the F_2 objective function.

The *Local Heaviest Approximation* algorithm avoids to access control volumes from the smallest to the largest *Aspect Ratio*, reducing the time complexity of the matching algorithm. The overall grid quality, however, is sacrificed, as it produced larger F_2, F_3 and F_4 results in most runs.

The *Path Growing Algorithm* produces better results in F_4 and F_2 in every run under these conditions. However, we can see some fluctuations on F_3. Notice that this algorithm produced better F_3 values in every serial run, and the variations took place whilst scaling the number of processors. It is worth to mention that it produced grids considerably larger, what can contribute to an increase in our quality metrics. This algorithm is slower than the Globular Matching algorithm when constructing grids in parallel, but it is faster when constructing them with serial runs.

To conclude, the *Minimum Approximation Matching* approach was considerably slower, obtaining less coarser grids. The execution times were expected to increase, since the time complexity of this algorithm is higher. In comparison to *Globular Matching*, this algorithm obtained grids with better overall quality, as we should keep in mind that we've obtained larger grids. When conducting a construction with this algorithm following F_3 + F_4, we obtain results in less time and with better quality, originated by the F_3 + F_4 combination.

Appendix A

Source Code

Local Heaviest Approximation

```
/**************************************************************************
* This function finds a match using the Local Heaviest Approximation Alg.
**************************************************************************/
void Match_Local_Heaviest_Approximation(CtrlType *ctrl, GraphType *graph)
   {
  int i, ii, k, j, dim, nvtxs, cnvtxs, mink;
  idxtype *xadj, *vwgt, *adjncy;
  idxtype *match, *cmap, *perm, *randperm;
  realtype *vvol, *vsurf, *adjwgt, *adjwgtsum, ar, minar;

  dim       = ctrl->dim;
  nvtxs     = graph->nvtxs;
  xadj      = graph->xadj;
  vwgt      = graph->vwgt;
  vvol      = graph->vvol;
  vsurf     = graph->vsurf;
  adjncy    = graph->adjncy;
  adjwgt    = graph->adjwgt;
```

```
adjwgtsum = graph->adjwgtsum;

cmap = graph->cmap = idxsmalloc(nvtxs, -1, "cmap");
match = idxsmalloc(nvtxs, -1, "match");

perm = idxsmalloc(nvtxs, -1, "perm");
randperm = idxmalloc(nvtxs, "randperm");
RandomPermute(nvtxs, randperm, 1);

/* insert the vertices according to the path growing algorithm */
ii = 0, cnvtxs = 0;
i = randperm[ii++];

while (cnvtxs <= .25*nvtxs && ii < nvtxs) {

   if (match[i] != UNMATCHED) {
    i = randperm[ii++];
    continue;
   }

   minar = DBL_MAX, mink = UNMATCHED;
   for (j=xadj[i]; j<xadj[i+1]; j++) {
      k = adjncy[j];
      if (k > i || vwgt[i] + vwgt[k] > ctrl->maxsize || match[k] !=
          UNMATCHED)
        continue;

      ar = ARATIO2(dim, vsurf[i]+vsurf[k]+adjwgtsum[i]+adjwgtsum[k]
                               -2.0*adjwgt[j], vvol[i]+vvol[k]);
      if (minar > ar) {
          minar = ar;
          mink = k;
        }
     }

     if (mink != UNMATCHED) {
       perm[cnvtxs] = i;
       perm[nvtxs-cnvtxs-1] = mink;
       cmap[i] = cmap[mink] = cnvtxs++;
       match[i] = mink;
       match[mink] = i;
     }
     i = randperm[ii++];
  }

  /* take care of the unmatched vertices */
  for (i=0; i<nvtxs; i++) {
     if (match[i] == UNMATCHED) {
       perm[cnvtxs] = i;
       cmap[i] = cnvtxs++;
       match[i] = i;
     }
  }

  CreateCoarseGraph(graph, cnvtxs, match, perm);

  IMfree((void**)&randperm, (void**)&perm, (void**)&match, LTERM);
}
```

Path Growing Algorithm

```
void choosePath(idxtype * perm, idxtype * cmap, idxtype * match, idxtype
    * seen, int * first,
               idxtype * patha, idxtype * pathb, int *sizea, int *sizeb,
               realtype *suma, realtype *sumb, int *alternate,
               int *cnvtxs, int *nvtxs, double * cost) {
  idxtype * path; int size, j;
  if ((*suma) > (*sumb)) {
    path = pathb, size = *sizeb;
    seen[(*first)] = UNMATCHED;
    *cost += (*sumb);
  } else {
    path = patha, size = *sizea;
    *cost += (*suma);
  }
  for (j = 0; j < size; j += 2) {
      int a = path[j], b = path[j + 1];
      perm[(*cnvtxs)] = a;
      perm[(*nvtxs)-(*cnvtxs)-1] = b;
      cmap[a] = cmap[b] = (*cnvtxs)++;
      match[a] = b;
      match[b] = a;
  }
  (*sizea) = 0, (*sizeb) = 0;
  (*suma) = 0.0, (*sumb) = 0.0;
  (*alternate) = 0;
}


/************************************************************************
```

```
* This function finds a match using the Path Growing Algorithm
**************************************************************************/
void Match_Path_Grow(CtrlType *ctrl, GraphType *graph) {
  int i, ii, k, j, dim, first, nvtxs, cnvtxs, sizea = 0, sizeb = 0,
      alternate = 0, mink;
  idxtype *xadj, *vwgt, *adjncy;
  idxtype *match, *seen, *cmap, *perm, *randperm, *patha, *pathb;
  realtype *vvol, *vsurf, *adjwgt, *adjwgtsum, suma = 0.0, sumb = 0.0,
      ar, minar;

  dim       = ctrl->dim;
  nvtxs     = graph->nvtxs;
  xadj      = graph->xadj;
  vwgt      = graph->vwgt;
  vvol      = graph->vvol;
  vsurf     = graph->vsurf;
  adjncy    = graph->adjncy;
  adjwgt    = graph->adjwgt;
  adjwgtsum = graph->adjwgtsum;

  cmap = graph->cmap = idxsmalloc(nvtxs, -1, "cmap");
  match = idxsmalloc(nvtxs, -1, "match");
  seen = idxsmalloc(nvtxs, -1, "seen");

  perm = idxsmalloc(nvtxs, -1, "perm");
  randperm = idxmalloc(nvtxs, "randperm");

  RandomPermute(nvtxs, randperm, 1);

  patha = idxmalloc(nvtxs, "patha");
```

```
pathb = idxmalloc(nvtxs, "pathb");

/* insert the vertices according to the path growing algorithm */
ii = 0, cnvtxs = 0;
i = randperm[ii++];
first = i;
double cost = 0.0;

while (ii < nvtxs) {

   if (seen[i] != UNMATCHED) {
        if (sizea > 0)
          choosePath(perm, cmap, match, seen, &first, patha, pathb,
              &sizea, &sizeb,
              &suma, &sumb, &alternate, &cnvtxs, &nvtxs, &cost);
        i = randperm[ii++];
        first = i;
        continue;
   }

   minar = DBL_MAX, mink = UNMATCHED;
   for (j=xadj[i]; j<xadj[i+1]; j++) {
      k = adjncy[j];
      if (k > i || vwgt[i] + vwgt[k] > ctrl->maxsize || seen[k] !=
          UNMATCHED)
        continue;

      ar = SQUAREDARATIO(dim, vsurf[i]+vsurf[k]+adjwgtsum[i]+adjwgtsum[k]
                                 -2.0*adjwgt[j], vvol[i]+vvol[k]);
      if (minar > ar) {
        minar = ar;
        mink = k;
      }
   }

   if (mink != UNMATCHED) {
     if (alternate) {
      pathb[sizeb++] = i;
      pathb[sizeb++] = mink;
      sumb += minar;
     } else {
      patha[sizea++] = i;
      patha[sizea++] = mink;
      suma += minar;
     }
     seen[i] = 1;
     alternate = (alternate + 1) % 2;
     i = mink;
   } else {
      if (sizea > 0)
          choosePath(perm, cmap, match, seen, &first, patha, pathb,
              &sizea, &sizeb,
              &suma, &sumb, &alternate, &cnvtxs, &nvtxs, &cost);
      i = randperm[ii++];
      first = i;
      continue;
   }
}

/* take care of the unmatched vertices */
for (i=0; i<nvtxs; i++) {
   if (match[i] == UNMATCHED) {
     perm[cnvtxs] = i;
     cmap[i] = cnvtxs++;
     match[i] = i;
   }
}

CreateCoarseGraph(graph, cnvtxs, match, perm);

IMfree((void**)&randperm, (void**)&perm, (void**)&match, (void**)&seen,
    (void**)&patha, (void**)&pathb, LTERM);
}
```

Minimum Approximation Matching

```
/************************************************************************
* This function finds a matching using the Match Minimum Approximation
************************************************************************/
void Match_Minimum_Approximation(CtrlType *ctrl, GraphType *graph)
{

  int i, ii, k, j, dim, nvtxs, cnvtxs, anvtxs, nedges;
  idxtype *xadj, *vwgt, *adjncy;
  idxtype *match, *cmap, *perm;
  realtype *vvol, *vsurf, *adjwgt, *adjwgtsum;
  FKeyValueType *edges;

  dim      = ctrl->dim;
```

```
nvtxs    = graph->nvtxs;
xadj     = graph->xadj;
vwgt     = graph->vwgt;
vvol     = graph->vvol;
vsurf    = graph->vsurf;
adjncy   = graph->adjncy;
adjwgt   = graph->adjwgt;
adjwgtsum = graph->adjwgtsum;
anvtxs = nvtxs;

perm = idxsmalloc(nvtxs, -1, "perm");

cmap = graph->cmap = idxsmalloc(nvtxs, -1, "cmap");
match = idxsmalloc(nvtxs, -1, "match");

if (anvtxs & 1) { anvtxs--; }

PerfectMatching pm(anvtxs, (xadj[anvtxs]/2));
pm.options.verbose = false;

edges = (FKeyValueType
   *)IMmalloc((xadj[anvtxs]/2)*sizeof(FKeyValueType), "edges");

nedges = 0;
for (i=0; i<anvtxs; i++) {

   for (j=xadj[i]; j<xadj[i+1]; j++) {
      k = adjncy[j];
      if (k >= i || k == anvtxs || i == anvtxs)
        continue;
      realtype ar = ARATIO2(dim,
          vsurf[i]+vsurf[k]+adjwgtsum[i]+adjwgtsum[k]
                                -2.0*adjwgt[j], vvol[i]+vvol[k]);
      pm.AddEdge(i, k, ar);

      edges[nedges].val1 = i;
      edges[nedges].val2 = k;
      edges[nedges].key = ar;
      edges[nedges].val = nedges;
      nedges++;
   }
}

pm.Solve();
ifkeysort(nedges, edges);
cnvtxs = 0;

for (ii = 0; ii < nedges && cnvtxs < .2*nvtxs; ii++) {
  i = edges[ii].val1, k = edges[ii].val2, j = edges[ii].val;
  if (pm.GetSolution(j)) {
    perm[cnvtxs] = i;
    perm[nvtxs-cnvtxs-1] = k;
    cmap[i] = cmap[k] = cnvtxs++;
    match[i] = k;
    match[k] = i;
  }
}

for (i=0; i<nvtxs; i++) {
   if (match[i] == UNMATCHED) {
      perm[cnvtxs] = i;
      cmap[i] = cnvtxs++;
      match[i] = i;
    }
  }

  CreateCoarseGraph(graph, cnvtxs, match, perm);

  IMfree((void**)&perm, (void**)&match, (void**)&edges, LTERM);
}
```

F4 macro definition

```
#define LIMITSQUARE 16
#define LIMITCUBE3 216


#define SQUAREDARATIO(dim, surf, vol) ((dim == 2) ?
    pow(((surf)*(surf)/(vol)) / LIMITSQUARE, 2) :
    ((surf)*(surf)*(surf))/((vol)*(LIMITCUBE3)))
```

K-Way refinement following F4

```
/***************************************************************************
* This function performs k-way refinement, whose objective is to directly
* minimize the sum of squared aspect ratios
***************************************************************************/
void Random_KWaySquaredARatioRefine(CtrlType *ctrl, GraphType *graph, int
    npasses)
```

```
{

  int i, ii, j, dim, nparts, pass, nvtxs, nmoves, ndegrees, pmax;
  int from, to, jbest, jbest1, jbest2;
  idxtype *xadj, *vwgt, *adjncy, *where, *pwgts, *perm, *phtable,
     *ptarget;
  realtype old, newar, best, best1, best2, id, ed, maxar;
  realtype OldToAR, NewToAR, OldFromAR, NewFromAR;
  realtype *vvol, *vsurf, *adjwgt, *pvol, *psurf, *degrees;

  nvtxs    = graph->nvtxs;
  xadj     = graph->xadj;
  vwgt     = graph->vwgt;
  vvol     = graph->vvol;
  vsurf    = graph->vsurf;
  adjncy   = graph->adjncy;
  adjwgt   = graph->adjwgt;

  where  = graph->where;
  pwgts  = graph->pwgts;
  pvol   = graph->pvol;
  psurf  = graph->psurf;

  dim     = ctrl->dim;
  nparts  = ctrl->nparts;
  degrees = realmalloc(nparts, "degrees");
  phtable = idxsmalloc(nparts, -1, "phtable");
  ptarget = idxsmalloc(nparts, -1, "ptarget");
  perm    = idxmalloc(nvtxs, "perm");
```

```
/* Determine the domain that has the maximum aspect ratio */
pmax = 0;
maxar = SQUAREDARATIO(dim, psurf[0], pvol[0]);
for (i=1; i<nparts; i++) {
   newar = SQUAREDARATIO(dim, psurf[i], pvol[i]);
   if (newar > maxar) {
     maxar = newar;
     pmax = i;
   }
}

IFSET(ctrl->dbglvl, DBG_REFINE,
   printf("Partitions: [%3d %3d]-[%3d %3d]. MaxRatio: [%4d, %e], Ratio:
      %e\n",
          pwgts[iamin(nparts, pwgts)], pwgts[iamax(nparts, pwgts)],
          ctrl->minsize, ctrl->maxsize, pmax, maxar, graph->minratio));

RandomPermute(nvtxs, perm, 1);

for (pass=0; pass<npasses; pass++) {
   RandomPermute(nvtxs, perm, 0);
   RandomPermute(nvtxs, perm, 0);

    for (nmoves=ii=0; ii<nvtxs; ii++) {
      i = perm[ii];
      from = where[i];

      if (pwgts[from] - vwgt[i] < ctrl->minsize)
        continue;
```

```
/* Determine the connectivity of the 'i' vertex */
for (id=ed=0.0, ndegrees=0, j=xadj[i]; j<xadj[i+1]; j++) {
  to = where[adjncy[j]];

  if (to == from)
    id += adjwgt[j];
  else
    ed += adjwgt[j];

  if (to != from && pwgts[to]+vwgt[i] <= ctrl->maxsize) {
    if (phtable[to] == -1) {
      degrees[ndegrees] = adjwgt[j];
      ptarget[ndegrees] = to;
      phtable[to] = ndegrees++;
    }
    else
      degrees[phtable[to]] += adjwgt[j];
  }
}

/* Determine which of the ndegrees moves is the best */
for (best1=0.01, best2=0.1, jbest1=-1, jbest2=-1, j=0; j<ndegrees;
    j++) {
  to = ptarget[j];
  OldFromAR = SQUAREDARATIO(dim, psurf[from], pvol[from]);
  OldToAR   = SQUAREDARATIO(dim, psurf[to], pvol[to]);
  NewFromAR = SQUAREDARATIO(dim, psurf[from]+id-ed-vsurf[i],
      pvol[from]-vvol[i]);
  NewToAR   = SQUAREDARATIO(dim,
      psurf[to]+ed+id-2.0*degrees[j]+vsurf[i], pvol[to]+vvol[i]);
```

```
/* Check first objective min(max) */

/* Check if it increases the max aspect ratio */
if (NewFromAR > maxar || NewToAR > maxar)
  continue;

/* If not... */
/* If move involves partition with max asp ratio, do the move
    now */
if (to == pmax || from == pmax) {
  jbest1 = j;
  break;
}

/* Else if partition with max asp ratio is not involved, do the
    move
   that gives best local gain */
else {
  old = amax(OldFromAR, OldToAR);
  newar = amax(NewFromAR, NewToAR);
  if (old-newar > best1) {
    best1 = old-newar;
    jbest1 = j;
  }
}

/* Check second objective squared AR */
old = OldFromAR + OldToAR;
newar = NewFromAR + NewToAR;
```

```
      if (best2 < old-newar) {
        best2 = old-newar;
        jbest2 = j;
      }
    }

    IFSET(ctrl->dbglvl, DBG_MOVEINFO,
          printf("\tjbest1=%d, jbest2=%d Gains: %8.6f %8.6f.\n",
              jbest1,
                 jbest2, best1, best2));

    if (jbest1 != -1) {
      jbest = jbest1;
      best = best1;
      IFSET(ctrl->dbglvl, DBG_MOVEINFO,
            printf("\t1st OBJECTIVE. Gain: %8.6f\n", best1));
    }
    else if (jbest2 != -1) {
      jbest = jbest2;
      best = best2;
      IFSET(ctrl->dbglvl, DBG_MOVEINFO,
            printf("\t2nd OBJECTIVE. Gain: %8.6f\n", best2));
    }
    else {
      jbest = -1;
      IFSET(ctrl->dbglvl, DBG_MOVEINFO,
            printf("\tNO OBJECTIVE. Gains: %8.6f %8.6f.\n" , best1,
                best2));
    }
    if (jbest != -1) {
      to = ptarget[jbest];

      where[i] = to;
      INC_DEC(pwgts[to], pwgts[from], vwgt[i]);
      INC_DEC(pvol[to], pvol[from], vvol[i]);
      psurf[from] = psurf[from] + id - ed - vsurf[i];
      psurf[to]  = psurf[to] + id + ed - 2.0*degrees[jbest] + vsurf[i];

      /* If we moved from/to the pmax subdomain find the newar one! */
      if (from == pmax || to == pmax) {
        pmax = 0;
        maxar = SQUAREDARATIO(dim, psurf[0], pvol[0]);
        for (i=1; i<nparts; i++) {
           if ((newar = SQUAREDARATIO(dim, psurf[i], pvol[i])) >
               maxar) {
             maxar = newar;
             pmax = i;
           }
        }

        graph->minratio = maxar;
      }
      nmoves++;

      IFSET(ctrl->dbglvl, DBG_MOVEINFO,
          printf("\tMoving %6d from %3d to %3d. Gain: %4.2f. MinRatio:
              %e [%e]\n"            , i, from, to, best,
              graph->minratio, vsurf[i]));
          /* CheckParams(ctrl, graph); */
        }

        for (j=0; j<ndegrees; j++)
          phtable[ptarget[j]] = -1;
      }

      IFSET(ctrl->dbglvl, DBG_REFINE,
            printf("\t[%6d %6d], Nmoves: %5d, MinRatio: %e\n",
            pwgts[iamin(nparts, pwgts)], pwgts[iamax(nparts, pwgts)],
            nmoves, graph->minratio));

      if (nmoves == 0)
        break;
    }

    graph->nmoves = nmoves;
    IFSET(ctrl->dbglvl, DBG_REFINE, printf("FinalMax: %d %e\n", pmax,
        maxar));

    IMfree((void**)&perm, (void**)&phtable, (void**)&degrees,
        (void**)&ptarget, LTERM);
}
```

References

[1] Numerical solution of weather and climate systems, November 2013.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] S. Buckeridge. Numerical solution of weather and climate systems. *International Journal for Numerical Methods in Fluids*, 21(3):783–805, 1995.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, February 2003.

[6] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69:125–130, 1965.

[7] Jack Edmonds. Paths, trees and flowers. *CANADIAN JOURNAL OF MATHEMATICS*, pages 449–467, 1965.

[8] M. Feldman. As moores law winds down, chipmakers consider the path forward. `https://www.top500.org/news/as-moores-law-winds-down-chipmakers-consider-the-path-forward/`, June 2018. Online; accessed 19 Jun 2018.

[9] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five Years of Electronic Design Automation*, 25 years of DAC, pages 241–247, New York, NY, USA, 1988. ACM.

[10] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[11] T. Fllenbach and K. Stben. *Algebraic Multigrid for Selected PDE Systems*, pages 399–410. 2011.

[12] Vipin Kumar George Karypis. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollel Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*, 1997.

[13] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[14] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[15] K.A. Hoffmann and S.T. Chiang. *Computational Fluid Dynamics*. Number v. 1 in Computational Fluid Dynamics. Engineering Education System, 2000.

[16] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *IPDPS*, pages 1–12. IEEE, 2010.

[17] W. L. Hoschv. Polynomial versus nondeterministic polynomial problem.

[18] G. Karypis I. Moulitsas. *MGridGen/ParMGridGen, Serial/Parallel Library for Generating Coarse Grids for Multigrid Methods*, 1st edition, December 2001.

[19] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[20] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[21] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998.

[22] Vladimir Kolmogorov. Blossom v: A new implementation of a minimum cost perfect matching algorithm.

[23] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[24] J. Van Lent. Multigrid methods for time-dependent partial differential equations. Technical report, Katholieke Universiteit Leuven, January 2006.

[25] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 242–255, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[27] Irene Moulitsas and George Karypis. Multilevel algorithms for generating coarse grids for multigrid methods. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 45–45, New York, NY, USA, 2001. ACM.

[28] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[29] David P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 225–231, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[30] T. Tezduyar. S. Aliabadi. Parallel fluid dynamics computations in aerospace applications, 1995.

[31] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *ESA*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011.

[32] R. Stanworth. Delta hpc cluster user guide. `https://intranet.cranfield.ac.uk/it/Documents3/Delta-UserGuide.pdf`, May 2018. Online; accessed 20 Jun 2018.

[33] Jack K. Steehler. Understanding moore's lawfour decades of innovation (david c. brock, ed.). *Journal of Chemical Education*, 84(8):1278, 2007.