Load Balancing for Parallel Coupled Simulations

Maria Predari

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Maria Predari**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Load Balancing for Parallel Coupled Simulations

Date de soutenance : 09 décembre 2016

Sous la direction de: Jean Roman

Laboratoire d'accueil: LaBRI, Inria

Devant la commission d'examen composée de :

Jean Roman	Professeur, Université de Bordeaux	Directeur de Thèse
Florent Duchaine .	Chercheur, Cerfacs	Examinateur
Rob Bisseling	Professeur, Utrecht University	Rapporteur
Umit Catalyurek	Professeur, Georgia Institute of Technology ...	Rapporteur
Pierre Manneback	Professeur, Université de Mons	Président du Jury
Aurélien Esnard ..	Maître de Conférences, Université de Bordeaux	Co-encadrant de Thèse

– 2016 –

Remerciements

First of all I would like to thank my supervisors Jean Roman and Aurélien Esnard for their support and guidance during the three years of this thesis. Both of them helped me in different ways and gave me the proper education for a potential future in the research. More precisely I would like to thank Aurelien for his enormous help during those years and his devotion to me and the subject. I would also like to thank the jury of my thesis and more precisely my two referees (Rob Bisseling and Umit Catalyurek) whose comments and reviews highly improved the quality of this manuscript. Also, I thank Florent Duchaine for his help on coupled simulations.

Following, I would like to thank my teammates at Hiepacs for the great times we shared together, our discussions and our common (and long) breaks. I would like to thank Abdou for his valuable advise and Meropi for her support. Finally, I thank my family for always being there for me and Emmanuel for his support, his solid help and for making my life happier.

Équilibrage de la charge des simulations parallèles couplées

Résumé

Dans le contexte du calcul scientifique, l'équilibrage de la charge est un problème crucial qui conditionne la performance des simulations numériques parallèles. L'objectif est de répartir la charge de travail entre un nombre de processeurs donné, afin de minimiser le temps global d'exécution. Une stratégie populaire pour résoudre ce problème consiste à modéliser la simulation à l'aide d'un graphe et à appliquer des algorithme de partitionnement.

En outre, les simulations numériques tendent à se complexifier, notamment en mixant plusieurs codes représentant des physiques différentes ou des échelles différentes. On parle alors de couplage de codes multi-physiques ou multi-échelles. Dans ce contexte, le problème de l'équilibrage de charge devient également plus difficile, car il ne s'agit plus d'équilibrer chacun des codes séparément, mais l'ensemble de ces codes pris dans leur globalité.

Dans ce travail, on propose de resoudre ce problème en utilisant le modèle de partitionnement à sommets fixes qui pourrait représenter efficacement les contraintes supplémentaires imposées par les codes couplés (*co-partitionnement*). Nous avons donc développé un algorithme direct de partitionnement de graphe qui gère des sommets fixes. L'algorithme a été implémenté dans le partitionneur Scotch et une série d'expériences ont été menées sur la collection des graphes DIMACS.

Ensuite nous avons proposé trois algorithmes de co-partitionnement qui respectent les contraintes issues des codes couplés respectifs. Nous avons egalement validé nos algorithmes par une étude expérimentale en comparant nos méthodes aux strategies actuelles sur des cas artificiels ainsi que sur des codes réels couplés.

Mots-clés

simulations numériques, simulations couplées, parallélisme, équilibrage de charge, partitionnement de graphe

Load Balancing for Parallel Coupled Simulations

Abstract

Load balancing is an important step conditioning the performance of parallel applications. The goal is to distribute roughly equal amounts of computational load across a number of processors, while minimising interprocessor communication. A common approach to model the problem is based on graph structures and graph partitioning algorithms.

Moreover, new challenges involve the simulation of more complex physical phenomena, where different parts of the computational domain exhibit different physical behavior. Such simulations follow the paradigm of multi-physics or multi-scale modeling approaches. Combining such different models in massively parallel computations is still a challenge to reach high performance. Additionally, traditional load balancing algorithms are often inadequate, and more sophisticated solutions should be explored.

In this thesis, we propose new graph partitioning algorithms that balance the load of such simulations, refered to as *co-partitioning*. We formulate this problem with the use of graph partitioning with initially fixed vertices which we believe represents efficiently the additional constraints of coupled simulations. We have therefore developed a direct algorithm for graph partitioning that manages successfully problems with fixed vertices. The algorithm is implemented inside Scotch partitioner and a series of experiments were carried out on the DIMACS graph collection. Moreover we proposed three co-partitioning algorithms that respect the constraints of the respective coupled codes. We finally validated our algorithms by an experimental study comparing our methods with current strategies on artificial cases and on real-life coupled simulations.

Keywords

numerical simulations, coupled simulations, parallelism, load balancing, graph partitioning

Équilibrage de la charge des simulations parallèles couplées

Résumé

Dans le contexte du calcul scientifique, l'équilibrage de charge est une étape importante conditionnant la performance des applications parallèles. L'objectif ici est de répartir la charge de calcul entre un nombre de processeurs donné afin de minimiser le temps d'exécution. Ceci est particulièrement critique lorsque le nombre de processeurs augmente, c'est-à-dire pour des applications hautement parallèles s'executant sur des architectures parallèles à plusieurs noeuds. Malheureusement, le problème ci-dessus est connu pour être NP-hard [30] mais de nombreux algorithmes d'équilibrage de charge existent, principalement basés sur des méthodes heuristiques.

Parmi eux, une approche commune pour résoudre l'équilibrage de charge d'une application consiste à modéliser le problème avec des structures à base de graphes (ou hypergraphes). Plus précisément, un sommet d'un graphe représente une tâche de calcul de la simulation tandis qu'une arête représente les dépendances entre les calculs. Par conséquent, pour distribuer la charge d'une simulation, on peut effectuer un partitionnement du graphe correspondant en k parties, chaque partie étant affectée à un processeur. Ainsi, le partitionnement de graphe apparaît comme une technique fondamentale pour la parallélisation des simulations, en minimiser le temps total d'exécution. Ils existent de nombreux outils de partitionnement du graphe tels que sont Metis [40], Zoltan [1] ou Scotch [53].

De nos jours, les simulations numériques deviennent de plus en plus complexes, mélangeant plusieurs modèles et codes pour représenter différentes physiques ou différentes échelles (multi-physics, multi-scale simulations). Ici, l'idée clé est de réutiliser les codes existants à travers un cadre (framework) de couplage, au lieu de les fusionner dans une application unique [2]. Ces simulations sont appelées simulations couplées. Un exemple typique de telles simulations est la modélisation du climat de la terre, qui implique au moins quatre codes pour l'atmosphère, l'océan, la surface terrestre et la glace [13]. La combinaison de ces différents codes est toujours un challenge difficile pour atteindre des performances élevées et pour passer à l'echelle.

Dans ce contexte, une question cruciale est sans aucun doute l'équilibrage de la charge des simulations couplées qui reste toujours une question ouverte. L'objectif ici est de trouver la meilleure distribution de données pour l'ensemble de l'application couplée et pas seulement pour chaque code pris séparément, comme c'est fait habituellement. En effet, l'équilibrage naïf de chaque code séparément peut conduire à un déséquilibre important et à des communica-

tions mal agençées entre les codes au cours de la phase de couplage. Cela peut considérablement diminuer la performance globale de l'application. Par conséquent, il est necessaire de modéliser le couplage lui-même afin d'assurer le passage à l'echelle, en particulier lors de l'exécution sur des milliers de processeurs. En d'autres termes, il faut développer de nouveaux algorithmes et de logiciel afin de appliquer un partitionnement " coupling-aware" de l'application entière.

De plus, des calculs récents (effectués au CERFACS[1]) pour des configurations industrielles concernant le problème du transfert de chaleur conjugué ont montré les avantages potentiels de la modélisation de ce problème avec des simulations couplées. Néanmoins, le temps total d'exécution est partiellement déterminé par la façon dont les maillages des deux solveurs sont partitionnés [26]. L'utilisation de nouveaux algorithmes pour équilibrer correctement la charge des simulations couplées avec un partitionnement de graphe adapté apparaît comme un moyen prometteur pour obtenir de meilleure performance des applications couplées. Nous nous attendons à ce que ce travail soit vraiment adapté à la prochaine génération de simulations scientifiques à grande échelle exécutées sur des architectures à plusieurs noeuds.

Nous présentons ci-après les principales contributions de ce travail. La première contribution implique la définition formelle du problème d'équilibrage de charge lorsque deux ou plusieurs modèles numériques (composants) sont couplés ensemble dans le cadre d'un phenomene physique plus complexe. Pour ce faire, nous présentons le modèle d'exécution des simulations couplées, qui évolue itérativement dans le temps en entrant de séquences de deux phases d'exécution différentes, régulières et couplées (séparées par étapes de synchronisation). Notez que pendant une phase régulière, chaque composant résout un système individuel définie dans son propre domaine de calcul, alors que pendant une phase de couplage, les composants interagissent principalement l'un avec l'autre, en échangant de données sur leurs interface de couplage.

Dans ce contexte, nous étendons la définition du partitionnement de graphe classique pour qu'il cible le problème de distribution de données des simulations couplées, ce que nous appellons le *problème de co-partitionnement*. De plus, nous proposons deux algorithmes qui éffectue un partitionnement "coupling-aware" plutôt que de diviser chaque composante de façon indépendante, c' est fait aujourd'hui dans de telles simulations. L'idée clé derrière les algorithmes de co-partitionnement est de prendre explicitement en considération l'existence de plusieurs composantes et leurs interactions potentielles. Par conséquent, des contraintes supplémentaires sont introduites dans le modèle et influencent les résultats de partitionnement entre différents composants ou différentes phases du même composant. Au cours de cette étude, nous validons nos algorithmes en termes de la qualité de partitionnement et les coûts de communication,

[1] http://cerfacs.fr

en mesurant le facteur de déséquilibre au cours des phases régulières et de couplage, la minimisation globale du coup d'arêtes coupées (edgecut) et le nombre de messages échangés entre les composants.

Nous évaluons initialement les algorithmes proposés sur des simulations et donées synthétiquement générés et ensuite nous testons des configurations réelles d' une simulation couplée utilisée dans le domaine de la propulsion aéronautique. Cette dernière étude a été réalisée en collaboration avec l'équipe CFD au CERFACS qui nous a fourni ce test. Plus précisément, deux différents solveurs qui représentent des propriétés physiques sont couplée pour simuler une chambre de combustion réaliste à l'intérieur d'une moteur d'hélicoptère. Les deux solveurs modèlent séparément la combustion et des phénomènes de conduction qui sont impliqués dans le processus, mais qui appartiennent des différentes propriétés telles que la discrétisation du maillage et les échelles de temps. Dans ce contexte, nous souhaitons évaluer les algorithes proposées ici du co-partitionnement sur les données ci-dessus en termes de la qualité de partitionnement et les coûts de communication. Les résultats obtenus font partie d'une étude préliminaire mais indiquent une direction prometteuse pour le problème de l'équilibrage de charge pour les simulations couplées.

La deuxième contribution de ce travail est liée à une variante du problème classique de partitionnement de graphe, c'est-à-dire le problème de partitionnement du graphe avec des sommets initialement fixes. Cette instance modifiée de partitionnement de graphe apparaît généralement lorsque l'application sous-jacente impose des contraintes supplémentaires sur le problème initial. Deux exemples qui utilisent le modèle ci-dessus sont l'équilibrage de charge des simulations adaptatives de raffinement de maillage [11] et la conception de circuit dans le contexte de VLSI [16]. La motivation de cette étude vient des contraintes supplémentaires imposées par les simulations couplées qui peuvent être modélisées en utilisant des sommets fixes supplémentaires dans le graphe sous-jacent.

Par conséquent, nous proposons un nouvel algorithme de partitionnement k -way qui utilise des techniques de croissance de graphe pour partitionner directement un graphe en k parties, sans utiliser le paradigme de la bissection récursive. Cette stratégie a été validée par des observations qui montrent que la bissection récursive produit souvent des partitions de moindre qualité lorsque les sommets fixes sont impliqués dans la procédure. L'algorithme est implémenté comme l'étape de partitionnement initiale d'un algorithme multiniveau à l'intérieur du cadre de Scotch partitionneur et une série des tests expérimentaux ont été réalisés en utilisant des graphiques de la collection bien connue *DIMACS'10* [7].

Le reste de ce travail est organisé comme suit: dans le chapitre 2, nous présentons l'état de l'art en termes de charge d'équilibrage et de couplage de codes. Plus précisément, nous citons brièvement les techniques de partitionnement de graphes bien connues et nous présentons les approches actuelles

pour coupler deux ou plusieurs simulations numériques. Dans le chapitre 3, nous presentons le problème de partitionnement de graphe avec des sommets fixes et nous proposons un algorithe (appelé KGGGP) qui résout le problème ci-dessus. Une vaste étude expérimentale sur la collection *DIMACS'10* est également inclus. Après, dans le chapitre 4, nous introduisons le problème de co-partitionnement avec une solution proposée basée sur de nouvelles techniques de partitionnement. De plus, on présente des résultats obtenus sur les cas de test générés par synthèse et sur une simulation couplée réelles fourni par CERFACS. Enfin, dans le chapitre 5, nous discutons nos conclusions concernant cette étude et nous donnons les prospectifs possibles pour les travaux futurs.

Contents

Chapter 1

Introduction

In the field of scientific computing, load balancing is an important step conditioning the performance of parallel applications. The goal is to distribute the computational load across multiple processors in order to minimize the execution time. This is especially critical when the number of processors increases, i.e., for highly parallel applications running on many-core architectures. Unfortunately the above problem is known to be NP-hard [30] but many load balancing algorithms exist, mainly based on heuristic methods. Among them, a common approach to solve the load balancing of an application is to model the problem with graph based structures and to apply graph partitioning techniques. Examples of mature and efficient partitioning tools are METIS [40], ZOLTAN [1] or SCOTCH [53].

Nowadays, numerical simulations are becoming more and more complex, mixing several models and codes to represent different physics or scales. Here, the key idea is to reuse available legacy codes through a coupling framework instead of merging them into a standalone application [2]. Such simulations are called *coupled simulations*. A typical example of such simulations is the modeling of the earth's climate that involves at least four codes for atmosphere, ocean, land surface and sea-ice [13]. Combining such different codes is still a challenge to reach high performance and scalability.

In this context, one crucial issue is undoubtedly the load balancing of the coupled simulation that remains an open question. The goal here is to find the best data distribution for the whole coupled application and not only for each standalone code, as it is usually done. Indeed, the naive balancing of each code on its own can lead to an important imbalance and to a communication bottleneck during the coupling phase that can dramatically decrease the overall performance. Therefore, one argues that it is required to model the coupling itself in order to ensure a good scalability, especially when running on thousands of processors. In other words, one must develop new algorithms and software implementation to perform a "coupling-aware" partitioning of the whole application.

Additionally, recent computations (at CERFACS[1]) for industrial configurations concerning the problem of conjugate heat transfer have shown the potential benefits of modeling this problem with coupled simulations. Nevertheless, the total execution time is partly driven by the way the meshes of the two solvers are partitioned [26]. The use of new algorithms to correctly load balance coupled simulations with enhanced graph partitioning techniques appears as a promising way to reach better performance of coupled applications on massively parallel computers.

Following, we present the main contributions of this work. The first contribution involves the formal definition of the load balancing problem when two or more numerical models (components) are coupled together as part of a more complex physical system. To do so, we introduce the execution model of coupled simulations, that evolves iteratively in time entering sequences of two different execution phases, a regular and a coupling one (separated by synchronization steps). Note that during a regular phase, each component solves an individual system defined in its own computational domain, while during a coupling phase, components mainly interact with each other, exchanging data on their overlapping domains. Under this context, we extend the definition of graph partitioning so that it addresses the data distribution problem of coupled simulations, denoted as the *co-partitioning problem*. Additionally, we propose two algorithms that perform a "coupling-aware" partitioning, instead of partitioning each component independently, as it is usually done in such simulations nowadays. The key idea behind the co-partitioning algorithms is to take explicitly into consideration the existence of several components and their potential interactions. Hence, additional constraints are introduced in the model and influence the partitioning results between different components or different phases of the same component. During this study, we validate our algorithms in terms of partitioning quality and communication costs, measuring the imbalance factor during regular and coupling phases, the global edgecut minimization and the number of messages exchanged among components.

We initially evaluate the proposed algorithms on synthetically generated mesh structures and then we test real-life configurations of a coupled simulation used in the field of aeronautic propulsion. The latter study has been done in collaboration with the CFD team at CERFACS that provided us with this test case. More precisely, two different solvers that represent distinct physical properties are coupled together to simulate a realistic combustion chamber inside a helicopter engine. The two solvers separately model the combustion and conduction phenomena that are involved in the process, but have very different properties such as mesh discretization and time scales. In this context, we are interested in evaluating the proposed co-partitioning algorithms on the above data in terms of partitioning quality and communication costs. The

[1] http://cerfacs.fr

results we obtain are part of a preliminary study but indicate a promising direction for the problem of load balancing for coupled simulations.

The second contribution of this work, is related to a variant of the classic graph partitioning problem, that is, the graph partitioning problem with initially fixed vertices. This modified instance of graph partitioning typically appears when the underlying application imposes additional constraints on the initial problem. Two examples that make use of the above model are the load balancing of adaptive mesh refinement simulations [11] and the circuit design in the context of VLSI [16]. Our motivation behind this study comes from the additional constraints imposed by the coupled simulations that may be modeled using additional fixed vertices in the underlying graph.

Therefore, we propose a new k-way partitioning algorithm that uses greedy graph growing techniques to directly partition a graph in k parts, without using the recursive bisection paradigm. This strategy has been validated by observations that show that recursive bisection often produces partitions of lower quality when the fixed vertices are involved in the procedure. The algorithm is implemented as the initial partitioning step of a multilevel algorithm inside the Scotch partitioning framework and a series of experimental tests have been conducted using graphs from the well-known *DIMACS'10* [7] collection.

The remainder of this work is organized as follows: in Chapter 2, we present the state-of-the-art in terms of load balancing and coupled simulations respectively. More precisely, we briefly review well-known graph partitioning techniques and we present current approaches to couple two or more numerical simulations. In Chapter 3, we discuss the problem of graph partitioning with fixed vertices and we propose a new graph partitioning method (named kgggp) that addresses the above problem. An extensive experimental study on the *DIMACS'10* collection is also included. Following, in Chapter 4, we introduce the co-partitioning problem along with a proposed solution based on new partitioning techniques. Moreover, experimental results on synthetically-generated and real-life test cases for CERFACS are presented. Finally, in Chapter 5, we discuss our conclusions regarding this study and we give possible prospectives for future work.

Chapter 2

Overview of the Problem and Related Work

In this chapter, we introduce the problem of load balancing for coupled simulations along with the methods that are currently employed to address it. Since load balancing in numerical simulations is highly associated with graph or hypergraph structures, we also present here the formulations and related techniques for graph partitioning.

2.1 Load Balancing in Scientific Computing

The increasing complexity of scientific computing often dictates a decomposition of the computational load in order to ensure high performance. In other words, when highly parallel simulations are executed on distributed environments of hundreds of processors, a distribution of the workload among the available processors is essential to achieve an efficient parallel solution. This distribution requires a data partitioning procedure and the use of distributed data structures in order to be performed successfully.

In applications with constant workload, a general strategy of distributing computations is to apply a static assignment of the workload to the processors at the beginning of the simulation. In other cases, where computations are unpredictable or change during execution, a redistribution of the workload is required at runtime. An example of such applications is the Adaptive Mesh Refinement (AMR) [11, 12] in which the mesh resolution is adapted dynamically within certain critical regions of the simulation.

Consequently, an equal distribution of a simulation's workload across the processors which, additionally, minimizes interprocessor communication may account for a significant reduction in the execution time of a parallel application. Note that communication costs are governed by the amount of data exchanged by the processors (communication volume) and the number of processors that share the same data (number of messages). The above problem

is defined as the *load balancing problem* and may highly determine the performance of parallel applications.

A plethora of load balancing strategies have been developed [9, 17, 23, 28, 48, 54, 57, 62], strongly governed by trade-offs between load balancing quality, the amount of data movement or load balancing speed. In general, the properties of an application determine which load balancing strategy should be used to decompose its workload. For instance, mesh-based PDE solvers and their sparse linear solvers mainly use graph-based partitioning algorithms [23, 28, 48, 57, 62] due to their excellent results on such applications. On the other hand, simpler geometric methods [9, 17, 54] are more suitable for applications such as particle simulations since they exploit available geometric information to obtain at a lower cost an efficient data decomposition. A great number of software libraries are available that provide high-quality implementations of partitioning procedures, some of which are described later in this work.

Even though existing partitioning methods have been very successful, research challenges remain and new partitioning algorithm should be proposed for the efficient execution of modern applications or new architectures. In this context, in the remainder of this document we will study the load balancing strategies of more complex, numerical simulations, that involve several models, each one representing different physics or scales. In such simulations, the corresponding meshes may have significantly different load characteristics depending on the different physics of the problem. Thus, traditional load balancing algorithms may be inadequate here, and more suitable data distribution techniques should be investigated.

Another example of modern challenges that however will not be addressed in this document is the resource-aware load balancing. Here, the hierarchical and heterogeneous nature of increasingly targeted cluster and grid architectures requires new techniques for load balancing. Algorithms that address this problem have been proposed for instance in [51] and [71].

2.2 Background on Graph Partitioning

Before we describe what is graph partitioning and how it is used to balance the load of parallel applications, we repeat here some well-known graph definitions and notations in order to keep this document self-contained.

2.2.1 Notations and Definitions

Definition 2.1 (Undirected Graph)**.**
A *graph* is a pair $G = (V, E)$ comprising a set of *vertices* V together with a set of edges E, such that $E \subseteq V \times V$. Moreover an undirected graph is a graph in which edges have no orientation. The edge (x, y) is identical to the edge (y, x),

i.e., they are not ordered pairs, but sets $\{x, y\}$ (or two-multisets) of vertices.

An *edge* (u, v) is said to be *incident* to vertices u and v, and accordingly u, v are said to be *adjacent* to edge (u, v). Moreover, when two vertices u, v are connected directly with an edge (u, v), u and v are said to be *neighbors*.

Definition 2.2 (Vertex degree)**.**
The *degree* of a vertex, denoted as $\deg(v)$, is the number of edges incident to the vertex. The maximum degree of a graph G, denoted as $\Delta(G)$, and the minimum degree of a graph, denoted as $\delta(G)$, are respectively the maximum and minimum degree of its vertices.

Note that a graph where each vertex has the same number of neighbors (same degree) is called a *regular* graph.

Definition 2.3 (Subgraph)**.**
A *subgraph* $S = (U, F)$ of a graph G is a new graph formed from G such that: U is a subset of V, and F is the restriction of E by U, such that $F = E \cap (U \times U)$

Definition 2.4 (Hypergraph)**.**
A *hypergraph* $H = (V, N)$ is a generalization of a graph where V is a set of vertices and N is a set of non-empty subsets of V called *nets* or *hyperedges*.

While graph edges are pairs of vertices, nets are arbitrary sets of vertices. The vertices in a net are called its *pins* and the number of pins of a net is called the *size* of it. Finally, it is often desirable to study hypergraphs where all nets have the same cardinality: a k-uniform hypergraph is a hypergraph such that all its nets have size k. So a two-uniform hypergraph is a graph, a 3-uniform hypergraph is a collection of unordered triples, and so on.

2.2.2 The Problem of Graph Partitioning

It is known that a common approach to solve the problem of load balancing is based on graph theory and more precisely on graph partitioning [33, 64]. In this context, a vertex of the graph represents a basic computational task of the problem (often related to a mesh element) and an edge represents a dependency in calculations between two tasks. Besides, each vertex has a weight proportional to the task's cost while each edge has a weight representing the communication cost between two computational tasks. Therefore, in order to balance the load of a parallel application among k available processors, one may perform a *graph partitioning* in k parts and assign each resulting part to a processor.

The main objective of graph partitioning is to divide the given graph into k smaller parts of roughly equal weight and minimize the number of edges cut between them as seen in Figure 2.1. Thus, when graph partitioning is used for

load balancing in a parallel environment, a balanced distribution of the workload among the available processors is obtained, while the communication costs of the application are minimized. Consecutively, graph partitioning appears as a fundamental technique for parallelization that offers high performance by substantially minimizing the total execution time.

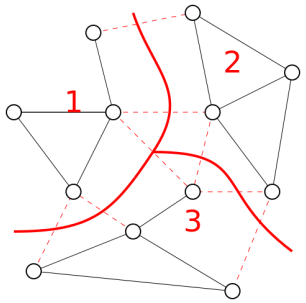


Figure 2.1: Example of a graph partitioned in three parts. We consider that the vertex and edge weights are equal to 1.

Let us consider a graph G where each vertex $u \in V$ has a weight $w(u)$ representing the computational load of a task and each edge $e = (u, v) \in E$ has a weight $w(e)$ representing the communication cost between tasks u and v.

Definition 2.5 (Graph partitioning in k parts)**.**
A partition of a graph G is a set of k vertex subsets $P = (V_1, V_2, \ldots, V_k)$, such that: each part $V_i, 1 \leq i \leq k$, is a non-empty subset of V; parts are pairwise disjoint ($V_i \cap V_j = \emptyset$ for all $1 \leq i < j \leq k$) and the union of the k parts is equal to V.

Given a vertex v mapped to the part V_i, we note $part[v] = i$ as its part number. Note also that when $k = 2$ the partition is called bipartition (or bisection) and the problem becomes the bi-partitioning problem.

Definition 2.6 (Problem of k-way graph partitioning)**.**

Given a graph G, the problem of k-way graph partitioning can be defined as the optimization problem of dividing a graph into k parts, $P = (V_1, V_2, \ldots, V_k)$, such that P:

i. is subject to the balance constraint

$$W_i \leq W_{avg}(1 + \epsilon) \text{ for } i = 1, \ldots, k; \tag{2.1}$$

ii. and optimizes the following objective

$$\min \sum_{e \in \mathcal{F}} (w(e)), \text{ where } \mathcal{F} = \{(a, b) \in E, a \in V_i \wedge b \in V_j \wedge i \neq j\} \tag{2.2}$$

In the above formula, W_i is the sum of the weights of all vertices in part V_i and $W_{avg} = \sum_{u_i \in V} w(u_i)/k$ represents the weight of each part in G under the perfect load balance. Moreover ϵ denotes the maximum imbalance tolerance as a percentage of the ideal weight where typical values are between $2\%, 5\%$, or 10%. Note that throughout this document ϵ will be set to 5% of the ideal weight, unless stated otherwise.

The objective in definition 2.2 is to minimize the *edgecut* metric $\sum_{e \in \mathcal{F}}(w(e))$ (where $\mathcal{F}$ is the edge separator). The edgecut of a partition is the weighted sum of all edges whose incident vertices belong to different parts. Another possible objective is to minimize the maximum *communication volume* metric [7]. For each vertex u, one determines the number of different parts $\mathcal{V}(u)$ in which u has neighbors, except $P[u]$. Then, the communication volume is given by the maximum over i, of the sum of all $\mathcal{V}$ for vertices u belonging to part i, i.e.

$$\max_{1 \leq i \leq k} \sum_{u \in V_i} \mathcal{V}(u) \tag{2.3}$$

In the remainder of this work, we use the edgecut metric to measure the quality of a partitioning since it is easy to compute; it is commonly used in the literature and it is known to successfully approximate the total communication volume of load balancing [32].

Accordingly, for hypergraph structures, the definition of a k-way hypergraph partitioning follows the one presented above, where a partition of a hypergraph is subjected to a balance constraint (defined in a similar way as for graphs) and optimizes a certain objective function. There are various objective functions that can be used as metrics for the quality of a valid hypergraph partitioning [19]. For instance, a commonly used metric that is shown to accurately model the total communication volume of parallel applications is called the *connectivity-1* metric. The objective function is defined as:

$$\min \sum_{n \in N} (w(n)(\lambda_n - 1)) \tag{2.4}$$

For a k-way partition P, a net that has at least one pin in a part is said to connect that part. The number of parts connected by a net n defines its *connectivity* and is denoted as λ_n. A net n is said to be internal if it connects exactly one part (i.e. $\lambda_n = 1$), and cut otherwise (i.e., $\lambda_n > 1$). Thus $w(n)(\lambda_n - 1)$ defines the cost of net n related to *connectivity-1*.

Finally, another problem which is highly related to graph partitioning and will be often mentioned throughout this manuscript is the *repartitioning* problem.

Definition 2.7 (Problem of k-way graph repartitioning)**.**
Given a weighted graph G and an unbalanced partition P in M parts, the classic repartitioning problem aims to compute a new partition P' of G (in M

parts) that satisfies the balance constraint and minimizes 1) the edgecut, and 2) the *migration volume*.

The additional objective aims to minimize the migration time, which consists of minimizing the communication costs required in order to redistribute the data among parts.

2.2.3 Graph Partitioning Algorithms

Unfortunately, the graph partitioning problem and its variants fall in the category of NP-hard problem [30], so finding a balanced partition of a given graph is usually based on approximation algorithms. Exact algorithms, where the solution space is extensively explored, exist (e.g. dynamic programming) but are too expensive in terms of execution time and will not be studied in this work.

Here, we briefly present some of the most common heuristic techniques for graph partitioning that are divided into two categories based on their view of the problem: the *local approach* (or refinement) and the *global approach*.

Global Approaches

Spectral Method. The spectral graph partitioning approach consists of finding the eigenvalues of the *Laplacian* matrix L associated with a given graph G. The Laplacian matrix is defined as $L = D - A$, where A is the adjacency matrix with A_{ij} implying an edge between node i and j, and D the diagonal matrix, where each entry D_{ii} represents the degree of the node i. Since L is a positive semi-definite matrix, its eigenvalues can be ordered as follows: $\lambda_1 = 0 \leq \lambda_2 \leq \ldots \leq \lambda_n$. The algebraic connectivity of a graph G is the second-smallest eigenvalue of the Laplacian matrix, and yields a lower bound on the optimal cost of a bipartitioning. This eigenvalue is greater than 0 if and only if G is a connected graph, which derives from the fact that the number of times 0 appears as an eigenvalue in the Laplacian is the number of connected components in the graph. The eigenvector v_2 corresponding to λ_2, called the Fiedler vector, bisects the graph into two parts based on the sign of the corresponding vector entry. Division into a larger number of parts is usually achieved by recursively applying the procedure to subgraphs. While this method results in partitions with good quality, it is very expensive in terms of execution time. Some graph partitioning algorithms that are based on the spectral information are found in [33, 48, 49, 67]

Bubble/Greedy Growing. The bubble growing algorithms is usually used as a global graph partitioning and in the most simple versions of this approach, like the algorithm of Farhat [28] or the greedy graph growing [8, 35], initial seeds are included to guide the growing development of parts. During the

partitioning procedure, the seeds are visited one after the other and so the parts are created sequentially based on the graph traversal. Since the bubble growing approach strongly depends on the selected seeds and usually produce solutions of poor quality for the lastly created parts, those methods call the partitioning routine multiple times using different initial seeds and the best partitioning result is chosen at the end.

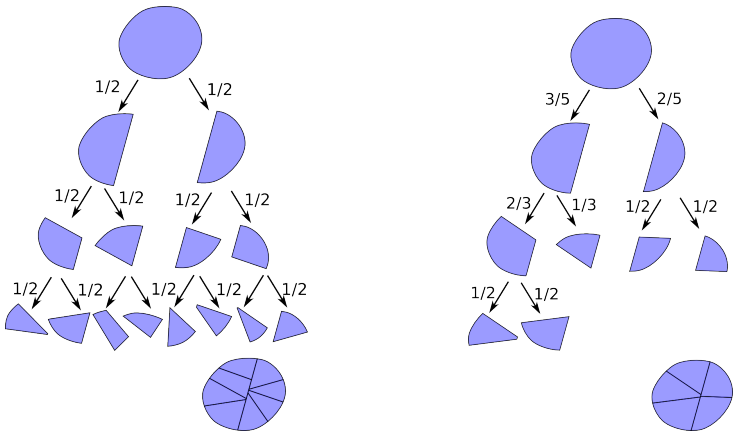
In most advanced versions of this approach the parts are managed and developed in a simultaneous way [50]. The main idea here is that the bubbles start to grow using the seeds as centers until they become stable and their union cover all the vertices in the graph. The partition is defined by considering the bubbles as the new parts and the center of those bubbles become the new seeds. The above procedure continues until the distance between the old and the new center of the bubble becomes small enough. Moreover, during the part growing procedure, the algorithm tries to minimize both the contact between different parts and the maximal vertex distance of a part to the bubble's center. Thus, the algorithm results in creating bubbles that are as much convex as possible, which lead to a more optimized partitioning considering the quality of the edgecut result.

Recursive Bisection. Nowadays the size of graphs deriving from parallel numerical simulations becomes larger and larger, thus the majority of modern graph partitioning methods follow a divide and conquer approach to solve the problem. The idea behind the recursive approach is to partition the graph in a number of small parts, usually two but it could be four or even eight, and to repeat the same process in each resulting part until the right number of parts is obtained. Since the number of parts is commonly set to two the recursive approach is called *recursive bisection*.

In Figure 2.2, one may see how a graph is partitioned in eight and five parts (in 4.3a and 4.3c respectively). As one may observe in Figure 4.3c, the graph can be easily partitioned in any number of parts k, even if it is not a power of two. The above is possible by forcing bisections that result in two unbalanced subgraphs during some recursive steps, as long as at the end of the overall procedure the proper number of balanced parts is obtained. Note that the recursive bisection can use any algorithm to compute each bisection, so different algorithms can be used to implement the actual bisection step. The recursive approach has been shown to be fast and to result in partitions of relative good quality; however some disadvantages of the method will be discussed later on this document and more specifically in chapter 3.

Local Approaches

The algorithms that are presented here are examples of the local graph partitioning approaches, that make no use of the global graph structural infor-



(a) in 8 parts (b) in 5 parts

Figure 2.2: Illustration of recursive bisection approach in three bisection levels.

mation, and locally improve a solution starting from an initially (random) bisection. Multiple trials of the above methods may increase the amont of local view as the number of trials increases.

Kernighan-Lin Algorithm. The Kernighan-Lin [43] (KL) algorithm is an iterative algorithm, performed in several passes, that starts from an initial bi-partition and repeatedly selects a pair of vertices to exchange parts. A vertex that has already changed parts will not be considered for another displacement during the same pass. The algorithm continues the vertex exchanges even if it temporarily deteriorates the edgecut that leads to exploring more solutions instead of remaining blocked in a locally optimal partition. At the end of a pass, only the displacements that led to the best edgecut improvement will be considered. The fact that vertices are exchanged in pairs helps in maintaining the balance constraint, however it restricts the edgecut minimization. Finally, multiple passes may be needed to further improve the edgecut, and the complexity of the algorithm is $O(|V|^2 \log(|V|))$.

Fidducia-Mattheyses Algorithm. The Fiduccia-Matthyses [29] (FM) algorithm is an improvement of the KL algorithm in terms of execution time, exhibiting an almost linear behavior that typically converges in several passes. Opposite to the KL algorithm, FM performs displacements of a vertex from one part to the other one instead of vertex pair exchanges. Naturally, this may lead to load imbalance between the parts, therefore only displacements that respect the balance constraint are allowed.

The FM algorithm maintains for each vertex a gain value which represents the variation in the total edgecut when the vertex is moved to the other part. The vertices are sorted based on their gain value into a bucket structure. This

is done using an array whose j^{th} entry contains a doubly-linked list of vertices with gain currently equal to j. Additionally, another array is maintained to access the elements of the linked lists by their vertex number allowing a quick search and update of the gain values. This data structure can be seen in Figure 2.3.

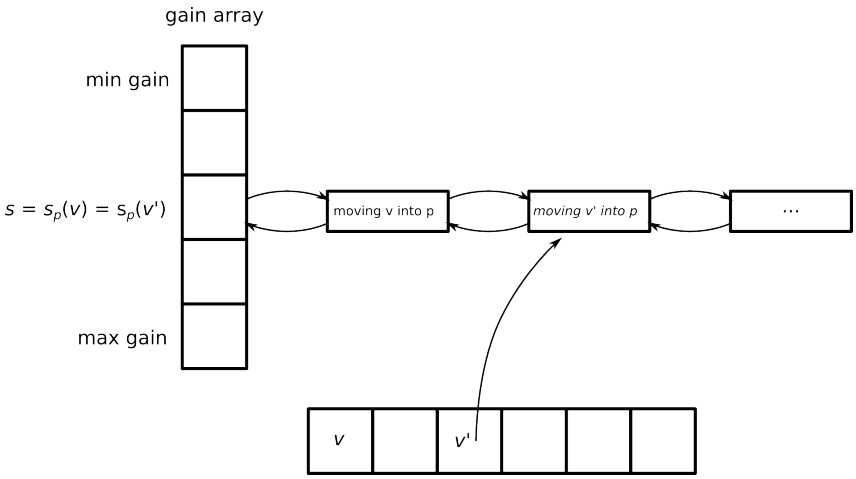


Figure 2.3: Gain bucket structure introduced in FM algorithm. $s_p(v)$ defines the gain function of moving vertex v into part p

Throughout the algorithm, a vertex that corresponds to the maximum gain value is chosen from the bucket structure to be moved to the other part. If this displacement does not violate the balance constraint, the algorithm performs the displacement and the vertex is marked as moved. Finally FM updates the gain values in the bucket structures for the neighbors of v that have not yet been moved. Employing this simple but efficient data structure FM finds the vertices with the maximum gain faster than KL in an almost linear time.

The main drawback of the iterative algorithms described above has to do with their strictly local vision of the problem which may produce solutions that are no better than a local optimum and strongly dependent on the initial partition. However as will be explained later on, such iterative algorithms are extensively used to refine roughly good partitioning solutions.

2.2.4 Multilevel Framework

In the early '90s, the multilevel recursive bisection algorithm (abbreviated as MLRB) emerged as a highly effective method for computing a k-way partitioning of a graph [14, 42, 48]. A major contribution of MLRB is the introduction of the multilevel framework which is widely used nowadays in most graph partitioning algorithms. The basic structure of the multilevel framework is rather simple. The main objective here is to reduce the size of a graph with

local optimizations, so that the solution space is reduced equally, resulting in a smaller problem where a partitioning algorithm could be applied in reasonable time. As we have mentioned before, the size of graphs (which model numerical simulations, VLSI design, etc) has become larger and larger and methods such as spectral partitioning can not be applied directly without increasing the runtime to a significant extent. Employing the multilevel framework, the size of a graph can be reduced without entirely losing its topological properties and the problem of graph partitioning can be solved much faster.

The multilevel framework consists of three phases:

i. *coarsening phase*, where a coarsening algorithm is performed recursively on the initial graph until a smaller structure (coarsest graph) is obtained;

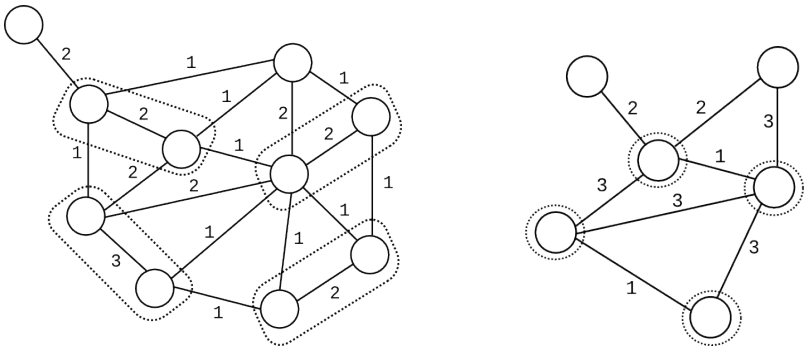
ii. *initial partitioning phase*, during which a partition of the coarsest graph is calculated;

iii. *uncoarsening phase*, where the partition obtained from the previous phase is projected back to the original graph. During this phase a refinement algorithm is also applied after each projection.

Coarsening Phase

During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$, is constructed from the original graph $G_1 = (V_1, E_1)$ such that $|V_i| < |V_{i-1}|$. In most coarsening schemes, a set of vertices of G_i is combined to form a single vertex of the next level coarser graph G_{i+1}, called *super vertex*. Adding up the weights of the combined set of vertices in G_i determines the weight of their corresponding super vertex v in G_{i+1}. The edge weight of v is calculated accordingly. This coarsening method ensures the following properties [48]: i) the edgecut of the partitioning in a coarser graph is equal to the edgecut of the same partition in the finer graph; ii) a balanced partition of a coarser graph leads to a balanced partition of the finer graph.

This edge collapsing idea can be formally defined in terms of matchings [2, 12] and more precisely maximal matching. A matching of a graph is a set of edges, no two of which are incident on the same vertex. A matching is called maximal, if it is not possible to add any other edge to it without making two edges become incident on the same vertex. The coarsening phase ends when the coarsest graph G_l has a small number of vertices or if the reduction in the size of successively coarser graphs becomes too small. Among many matching algorithms like RM (random matching) [48], LEM (light edge matching) [14] and HEM* [42], here we briefly describe the HEM (Heavy Edge Matching) algorithm which is widely used in modern partitioning tools. The HEM algorithm works as follows: the vertices are visited in a random order and each vertex u is matched with one of its adjacent unmatched vertices v, such that



(a) Vertex pairs of G_i coarsening level.

(b) Vertices of G_{i+1} level after contraction.

Figure 2.4: Contraction of egdes in HEM. Note how the edge weights are updated after the contraction of vertices. Vertex weights are not illustrated

the weight of the edge (u, v) is maximal over all possible incident edges (Figure 2.4). Hence, the HEM algorithm naturally encourages the minimization of edgecut.

Initial Partitioning Phase

The initial partitioning phase of the multilevel framework aims to calculate a partitioning of the coarsest graph G_l that will later be projected back to the initial graph structure. Since the quality of the partition of G_l will highly influence the quality of the final partition, the choice of the partitioning strategy in this step is important. Note that any partitioning strategy mentioned above, may be used during this step; among them the spectral approach, greedy algorithms, or even another multilevel algorithm. Since G_l is small in size compared to G_1, the multilevel framework tends to yield partitions of good quality, since a more global view of the problem is possible, and later tries to preserve them during the uncoarsening phase.

Uncoarsening Phase

The uncoarsening phase is the last step of the multilevel framework and the main objective here is to project the partition obtained on the coarsest graph G_l back to the initial graph G_1 going through the sequence $G_l, G_{l-1}, \ldots, G_2, G_1$. Since a partition P_i is locally optimized for graph G_i and not necessarily for the graph of the next level G_{i+1}, a refining procedure should be applied in order to preserve the quality of each P_i and eventually the quality of the very first partition P_1. It is important to observe that graph G_{i-1} has more degrees of freedom compared to G_i, since it unveils new vertices that may have an influence on the quality of the partition in this level. Hence local refinement algorithms are used after each projection step and are usually based on KL or

FM algorithms to further improve the partition.

Remember that KL or FM are iterative algorithms that have a local view of the partitioning problem and starting from a bisection, they swap vertices among the two parts in order to reduce the edgecut of the partition. Those methods along with their extensions to k-way refinement, that is refinement algorithms directly for k parts and not just for a bisection, are extensively used during the uncoarsening phase.

2.2.5 Multilevel Algorithms: MLRB and MLKW

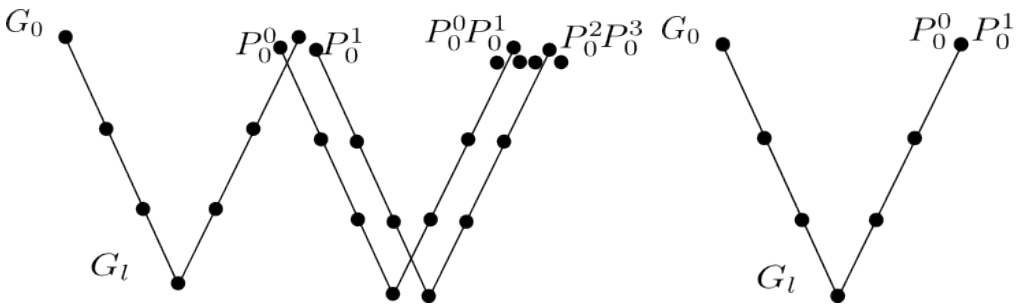
As mentioned before, the first method that used the multilevel scheme is the MLRB algorithm which combines the multilevel framework with the classic recursive bisection in order to obtain a k-way partition. MLRB follows the W cycle paradigm seen in 2.5a where more that one calls to the multilevel framework are performed. Starting from the original graph, a call to the multilevel framework is made which results in a refined bipartition. That is, during the initial partitioning phase the coarsest graph is partitioned in two parts with a bisection algorithm and then it is refined using the FM algorithm. Consecutively, for each resulting part a new multilevel framework is performed and this procedure continues recursively until the desired number of parts is obtained. As we may see in Figure 2.5a, a 4-way partition is obtained with 3 calls to the multilevel framework. The complexity of the MLRB for producing a k-way partitioning of a graph G is $O(|E|\log(k))$.

Moreover, the multilevel paradigm can also be used to construct a k-way partitioning of the graph directly during the initial partitioning phase as illustrated in 2.6. This method follows the V-cycle paradigm shown in 2.5b. In a V-cycle the multilevel framework is called just once. Again, the graph is coarsened successively to a smaller structure, but the coarsest graph is now directly partitioned into k parts. During the uncoarsening phase, this k-way partition is refined successively as the graph is projected back into the original graph.

There are a number of advantages for computing the k-way partitioning directly (rather than computing it successively via MLRB). First, the entire graph now needs to be coarsened only once, reducing the complexity of this phase to $O(|E|)$ from $O(|E|\log(k))$. Secondly, it is well known that recursive bisection can do arbitrarily worse than k-way partitioning [63]. Thus, a method that obtains a k-way partitioning directly can potentially produce partitions of better quality. Note that the direct computation of a good k-way partitioning is harder than the computation of a good bisection in general (although both problems are NP-hard). This is because the problem of finding the best assignment of a vertex to a part among k possible parts such that the edgecut is minimized, is more complicated than choosing between just two parts. In the former case, the optimization space is increased combinatorially.

However in the context of multilevel framework, we only need a rough k-way partitioning of the coarsest graph, as this can be potentially refined as the graph is uncoarsened. The coarsest graph can be partitioned in k parts using any known algorithm, for instance a method is to simply coarsen the graph down to k vertices. During the refinement phase, one needs to refine a k-way partitioning, which is considerably more complicated than refining a bisection. However, algorithms that refine a direct k-way partition exist as for example in [58] where a generalization of FM algorithm is proposed for refining a k-way partition with a complexity of $O(|E|)$.

Overall methods that follow the above approach are called MLKW (for multilevel k-way partitioning) and the best known algorithms have a time complexity of $O(|E|)$ and produce partitions that are comparable or of better quality that MLRB in substantially less time.



(a) W cycle scheme. (b) V cycle scheme.

Figure 2.5: Illustration of the multilevel framework.

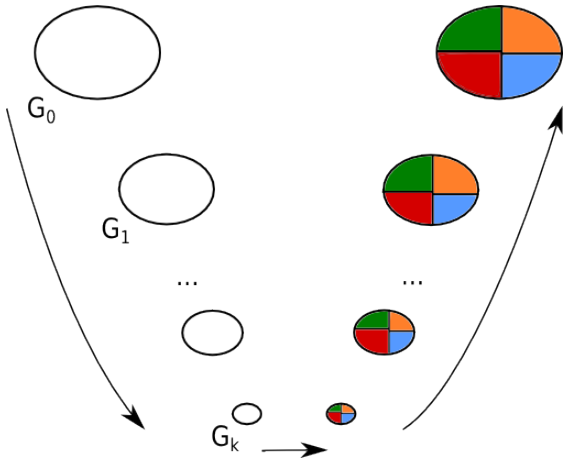


Figure 2.6: Representation of a 4-way direct partitioning using the multilevel framework.

In the literature many integrated software for graph or hypergraph partitioning exist, that often use MLRB or MLKW such as SCOTCH, KMETIS or

PaToH (a detailed list of tools is presented in Chapter 3). Finally, there are some partitioning tools that use parallel algorithms like ParMetis [40], which extends its functionalities under an MPI-based parallel environment, and PT-Scotch [53] that proposes a multi-threaded and multi-processor graph partitioning.

2.3 Coupled Simulations

2.3.1 Introduction

A new challenge emerging in scientific computing involves the understanding and systematic modeling of more complex physical phenomena that are found in many academic fields such as plasma physics, climate and weather, hydrology and material science.

In such complex systems, several phenomena are considered at the same time which results in an increased complexity of the underlying numerical models and tools. The reason is that each phenomenon is described by at least one mathematical equation and these equations rarely have the same mathematical properties.

One approach to address such problems is to solve all different equations at once, thus creating a standalone simulation, ensuring the coupling between different physical equations. However, even though this approach ensures the continuity of all variables, it has a few drawbacks. First of all, as mentioned above, different equations may require different numerical methods. It can become quite difficult to implement all of these methods in one code and to solve all of them together in an efficient way.

Secondly, the configurations to be studied are often unsteady, adding to the global complexity. The unsteady nature leads to another issue: the different phenomena have different time scales. For instance, when comparing the heat conduction in a wall with a reactive flow, the time scale of the former model is often two to three orders of magnitude higher than the time scale of the latter. Hence, solving both equations at once, implies using the same integration time step which is limited by the fastest physics, leading to a waste of computational resources.

Finally, an important disadvantage of using standalone simulations to model such systems is that re-usability of existing models is not possible. In other words, legacy codes cannot be integrated directly inside the simulation, instead, they have to be re-implemented in order to be compatible with the code. Additionally, standalone simulations may be hard to maintain. For instance, if at some point a better tool is available for a certain phenomenon, it cannot easily replace an existing one without having to modify the overall simulation.

Given the above reasons, an efficient approach to address such complex phenomena is to develop modern simulations that combine distinct numerical models, plugged in together under a coupling framework. In this section, we give an overview of such simulations and we refer to them as *coupled simulations*.

Examples of coupled simulations often follow the multi-physics or multi-scale modeling approaches. To simulate real-word conditions under the multi-physics model, one must consider the impact of a number of different physics that occur concurrently. Typically, in such simulations, effects from one physical phenomenon influence the behavior of an object in the computational domain of another physical phenomenon. Understanding multi-physics behavior is therefore a major challenge to accurately predict the performance of such simulations. In addition, a broad range of scientific problems involve multiple scales. Indeed in applications such as the simulation of crack propagation [4], traditional mono-scale approaches have proven to be inadequate, even if a parallel machine is used. This is due to the range of scales and the prohibitively large number of variables being involved in the simulation. Thus, there is a growing need to develop systematic modeling and simulation approaches for multi-scale problems.

Consequently, the potential benefits of using coupled simulations are more important than those of using standalone simulations. However different challenges still exist. Indeed, combining such diverse numerical models in massively parallel coupled simulations, is not an easy task, neither in terms of the architecture of the coupled simulation nor in terms of data processing during the coupling.

To support the successful execution of coupled simulations, software tools (called *couplers*) have been developed and provide the necessary functionalities to plug in together multiple standalone solvers (called *components* in the following). In general, coupling frameworks are responsible for two main operations: to manage the execution of the components within a coupled simulation and to perform the exchange of information at the coupling interfaces. For the first one, coupling frameworks need to specify the sequencing, synchronization and coupling frequency among components while for the second one, they need to manage the communication infrastructure. Note that different components may have completely different internal characteristics (e.g. space resolution, domain decomposition etc) making the exchange of data more complicated. Therefore, the coupling framework performs the data transfers between components and provide all the necessary coupling facilities such as mapping between different component grids (interpolation) or computing fluxes between components. Depending on the overall design of the coupling architecture, the above functionalities may be performed at different levels of the coupling framework. Some designs may include both a high-level driver and a coupler, whereas others may only include a coupler or may perform direct coupling

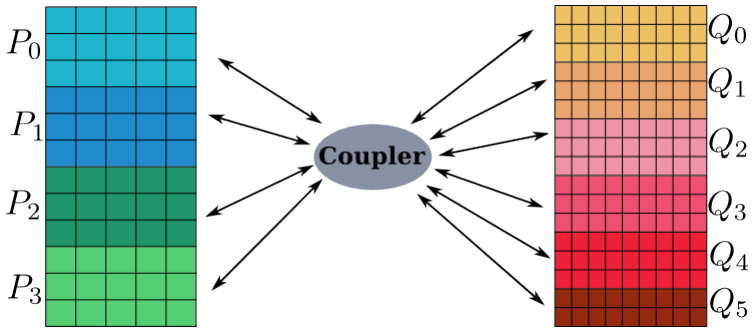
from within the components using a dedicated coupling library.

We may generalize the above coupling designs into two main categories, the couplers that have a *centralized communication scheme* (CCS) and the ones that follow a fully *distributed communication scheme* (DCS). The CCS is the most straightforward scheme that follows a many-to-one and one-to-many communication design as it is shown in Figure 2.7a. In this scheme, the coupler is sequential and, thus, runs on one processor. Furthermore, a variant of the CCS is shown in Figure 2.7b that employs a parallel coupler running on a set of processors $\mathcal{S}$. On the other hand, the DCS corresponds to a fully distributed, direct communication scheme, where only one level of communication is needed, enabled directly inside the components. In Figure 2.7c, we illustrate the above scheme, where one may see that the coupling processes are performed directly by the processor sets assigned to each component ($\mathcal{P}$ and $\mathcal{Q}$ respectively).

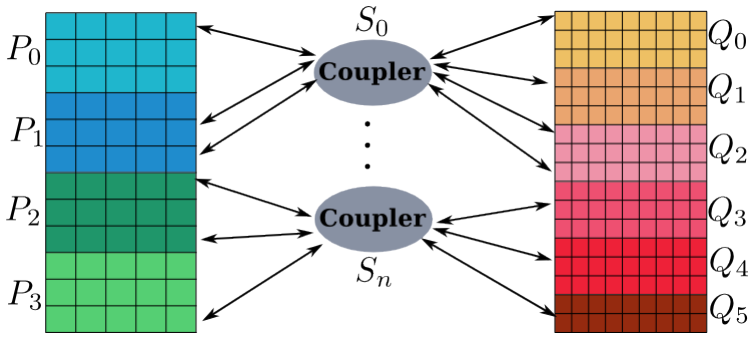
Some examples of parallel couplers that follow the CCS are the CPL7 [22] coupler, found in the latest version of the Community Climate System Model [13] (CCSM4) and the coupler of the Parallel Climate Model (PCM) [73]. More precisely, the coupler of the CCSM4 (CPL7) runs on a distinct subset of all the processors as if it was a separate component and thus performs the coupling operations in parallel. In this version of the CCSM4 the sequencing and hub attributes are migrated in a top-level driver. Finally, note that both couplers mentioned above, offer ad-hoc solutions designed for the specific problem of climate modeling. On the other hand, examples that follow the DCS are the OpenPALM [2], the MCT [46] and the MPCCI [37] coupling frameworks which are more suitable for generic solvers. Finally, OASIS [66] is another generic coupler that supports both CCS and DCS using the MCT coupling library (in the OASIS3-MCT version).

2.3.2 Examples

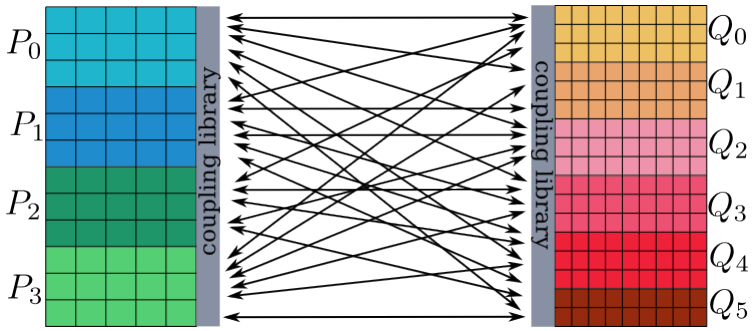
Understanding the global climate, how it has changed in the past centuries and how it is likely to change in the future, is a very important challenge that requires the systematic development of high-end climate models. Climate modeling is a typical example of coupled simulations following the multi-physics and multi-scale paradigm that requires huge amounts of computational power and may run for long periods of time. Nowadays, many climate models exist that simulate the earth's climate, including the CCSM4, the Coupled General Circulation Models (CGCMs) [61] or the PCM. Climate modeling is generally implemented as a coupled simulation and has been developed by different research groups for over decades, resulting in several software tools. These models usually consist of a coupling framework and four fundamental physical components: an atmosphere model, a surface land model, an ocean model, and a sea-ice model.



(a) CCS (sequential coupler).



(b) CCS (parallel coupler on a set $\mathcal{S}$ of processors).



(c) DCS.

Figure 2.7: Different communication schemes for coupling component models with different mesh resolutions and different partitions on sets $\mathcal{P}, \mathcal{Q}$ respectively.

Recent versions of the climate models have the ability to better support new science adding new components to the basic model such as atmospheric chemistry models or new land-ice components. As explained before, this is an advantage of the coupled simulation paradigm compared to a standalone simulation since it allows the re-usability of legacy codes with minimum modifications. Furthermore, following the coupled paradigm, individual models continue to be usable as single executables in the research groups where they have been initially developed, an important requirement in the climate modeling community.

To improve the understanding of the climate processes, the components are coupled by exchanging quantities in a consistent way across typical surfaces. Examples are the flux of energy and fresh water. This is a natural approach, because each component can be considered to evolve independently, while at the same time it is driven by the interactions with other components at the coupling interfaces. In other words, components periodically exchange two-dimensional boundary data, communicating via the coupling framework that is responsible for remapping the boundary-exchange data in space and time. In Figure 2.8 we illustrate an example of coupling between two components with different space resolutions, the atmospheric and the ocean one, performed by a coupler that follows the CCS design.

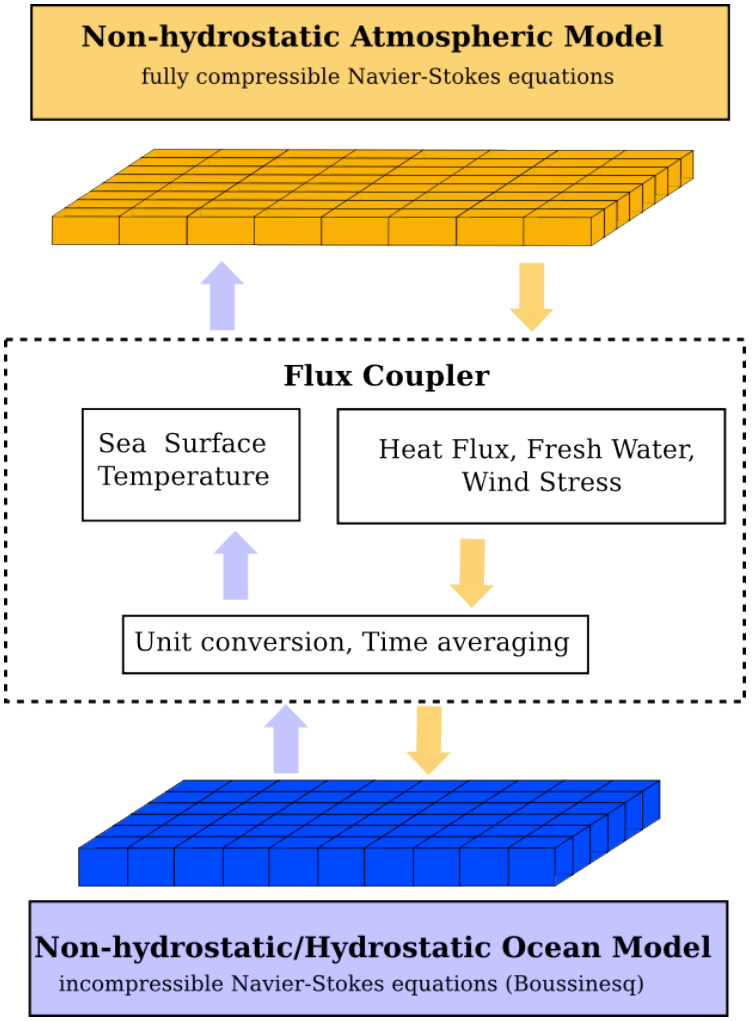


Figure 2.8: Coupling between atmospheric and ocean model.

Nevertheless, the accurate simulation of the global climate at high resolution and at a reasonable time is still a challenge despite the current advances

in climate modeling. This is mainly due to technological or financial limitations. Under this context, as stated in [24], the next generation of climate model should have the ability to significantly increase the spatial resolution, to include new components, and to improve the representations of sub-grid scale processes in order to produce regionally improved simulations of climate.

Another example of coupled simulations which appears in the field of computational biology, aims to model the highly complex process of thrombus formation in intra-cranial aneurysms[1] Note that thrombosis is a problem that depends on a large number of interactive biochemical agents and until today, there is no full understanding of the detailed process that leads to its creation. More precisely, thrombosis requires the concurrent simulation of both blood flow in complex geometries and biological processes such as blood coagulation under the influence of local flow properties. The main difficulty when coupling the above components is that they occur in highly different spatial and temporal scales, which prevents the application of classic simulation techniques and imposes the use of a multi-scale approach.

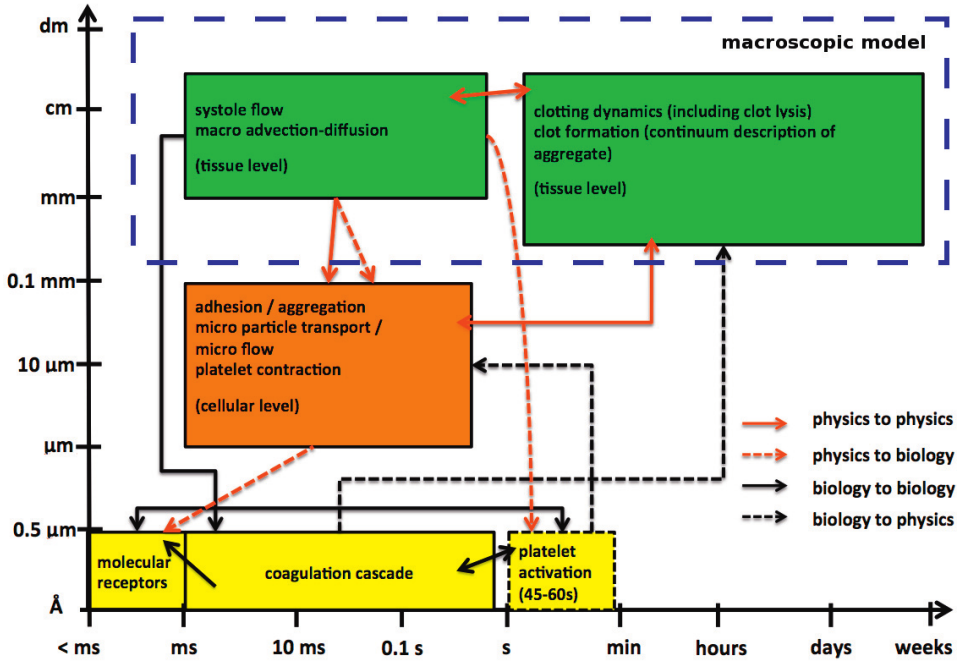


Figure 2.9: Reduced separation map of the spatial and temporal scales, (extracted from [76]).

In Figure 2.9, we illustrate the scale separation map that is typically used to describe the formation of a thrombus in intra-cranial aneurysms. A scale separation map is defined as a two dimensional map with the temporal scales

[1] An intra-cranial aneurysm is a cerebrovascular disorder in which weakness in the wall of a cerebral artery causes a localized dilation or ballooning of the blood vessel (thrombosis).

on the x- and the spatial scales on the y-axis. Here, we illustrate the map of a reduced complexity as it is presented in [76], since a full implementation of a multi-scale simulation for the thrombus formation is beyond reach at the moment. The simplified version presents different components of the coupled simulation as well as their spatial and temporal scales spanning over several orders of magnitude. In Figure 2.10, one may see the required interactions among the components and the data that are exchanged during the simulation of a thrombus formation.

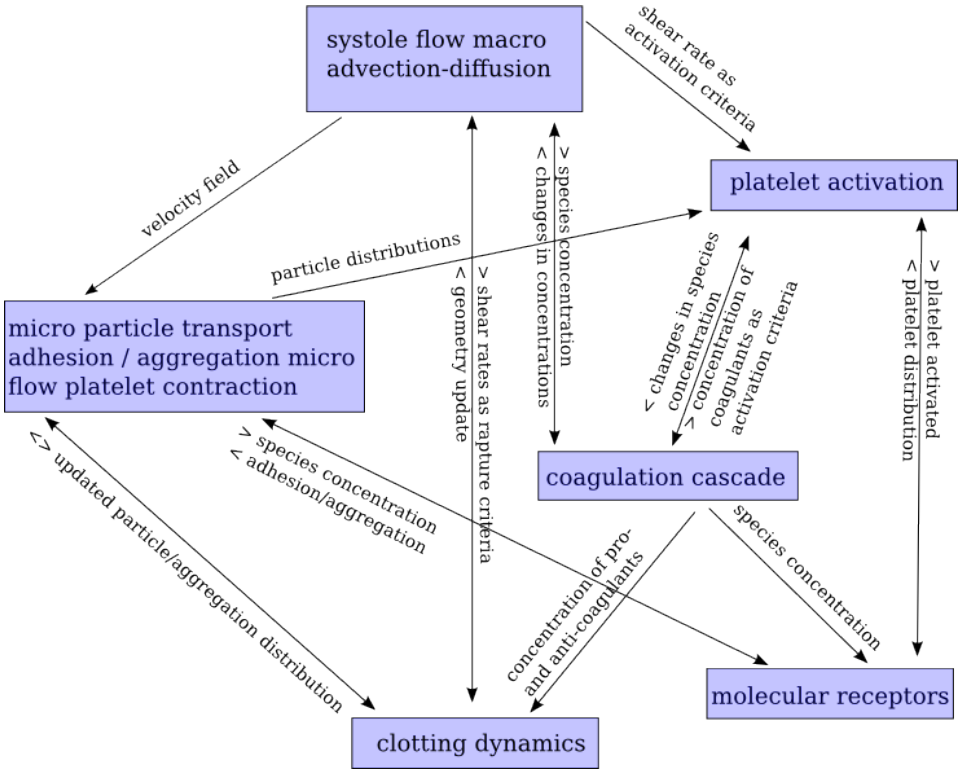


Figure 2.10: Interaction scheme with the data dependencies among components, (extracted from [76]).

Finally the authors of [76] state that the results obtained from the multi-scale simulation match in quality the expected process of thrombus growth after comparing them with clinical observations. The use of the multi-scale modeling approach for such a complex problem is highly crucial since without considering this solution, the simulation would not be feasible even on very large super-computers. It is important to note that the authors believe that the use of a distributed coupling framework (e.g. MAPPER [2]) might be extremely helpful to achieve the ambitious task of a realistic simulation of thrombus formation.

[2]http://www.mapper-project.eu

Other examples of coupled simulations can be found in the astrophysics domain where highly complex problems like the evolution of planetary systems, dense stellar clusters, and galactic nuclei can be simulated as multi-physics or multi-scale processes. For instance, in order to accurately model the formation of stellar clusters, the simulation should include a large range of physical components such as self-gravity, supersonic turbulence, hydrodynamics, outflows, radiation and magnetic fields. Also most astronomic problems are generally multi-scale with different spatial and temporal scales. For instance time may range from 10^4 minutes and 10^{-3} seconds to 10^{20} minutes and 10^{17} seconds. In [56] the authors present a software framework for combining existing computational tools for different astrophysical domains into a single multi-physics, multi-scale application. MUSE facilitates the coupling of existing codes written in different languages by providing inter-language tools and by specifying an interface between each module, as seen in Figure 2.11.

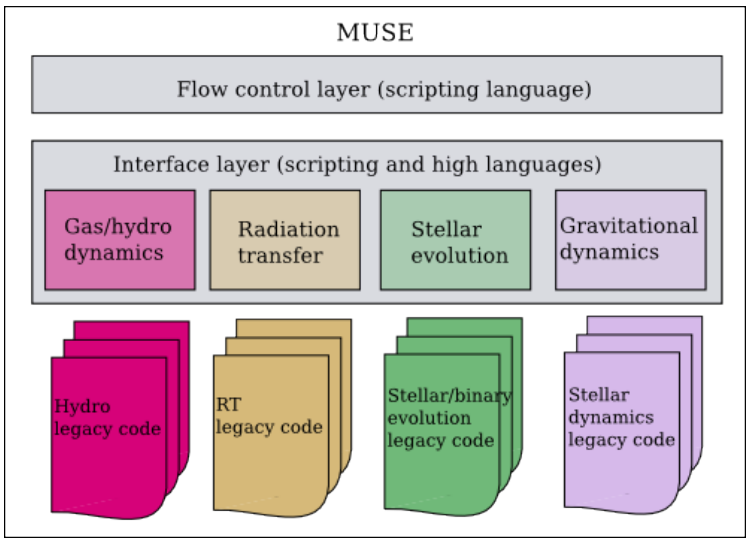


Figure 2.11: Basic structure design of the framework MUSE used for astrophysical applications (extracted from [56]).

Finally, another example of coupled simulations, that will be later reviewed in more details (in Chapter 4), emerges in the field of aeronautic propulsion where the behavior of hot components is impacted by complex interactions between different physics such as turbulent combustion, radiation and heat conduction [3].

2.3.3 Load Balancing

Reaching high performance within a coupled simulation is a challenging task since it is constrained by many factors such as the size of the problem, its complexity and its multi-component nature. Furthermore, within each component

there is an *internal imbalance* that contributes in the overall performance of the coupled simulation. In this context, the term internal imbalance refers to the load imbalance of a component as if it was executed in a standalone simulation. As expected, the internal imbalance of a component depends on the underlying domain decomposition and is governed by internal communication costs (among the processors of the same component).

As mentioned before, a coupling framework provides two main functionalities; it enables the coupling between different components and it facilitates their execution throughout the coupled simulation. Regarding the latter, most coupling frameworks improve the load balancing of the coupled simulation through efficient sequencing of components.

In a standalone simulation, different computational tasks may run in parallel on an independent set of processors to achieve high performance. In practice, this is more difficult to achieve within a coupled simulation, since coupled simulations are governed by scientific requirements that impose limitations to the coupling. For instance, in the coupled simulation of earth's climate, the atmosphere component should wait for the ice and land components to finish before it starts executing. Because of that, existing sequencing techniques for coupled simulations are mainly not generic and highly depend on the underlying problem. Finally, since couplings invoke a large number of data dependencies among components, communication costs are often important and not easy to minimize. Nevertheless, efficient scheduling techniques exist and contribute in the minimization of the total execution time.

Earlier, we divided the architecture of coupling frameworks into two categories, based on their underlying communication scheme CCS (sequential or parallel) and DCS. Following, we give an overview of the load balancing issues as they appear in each category.

Let us start considering a coupled simulation with two parallel components (C_0, C_1) and a sequential coupler that follows the CCS. An example of such a sequential coupler is OASIS3[3]. Before presenting the load balancing problem, it is important to explain the operations that take place during coupling and demonstrate the additional overhead. For this reason, we assume that the components are executed sequentially in time in respect to one another and in Figure 2.12 we depict one iteration of this execution. Note that the sequential (in time) execution is not preferred in real coupled simulations but sometimes, constraints on the sequencing of certain components may impose it. Finally, we consider that coupling happens bidirectionally, from C_0 to C_1 and from C_1 to C_0. Therefore, the operations that take place during one coupling are the following: both components send to the coupler at different times their coupling interfaces (at t_0 and τ_0 for C_0 and C_1 respectively). Then the coupler gathers the data from the interfaces, performs any essential computations such

[3]Do not confuse with the OASIS3-MCT or OASIS4 versions that uses the DCS.

as flux calculations or interpolations (at t_1 and τ_1) and finally sends the data to the second component (at t_2 and τ_2). As a result, the coupling overhead accounts for the exchange of data among different components (inter-component communication) and the computations performed by the coupler.

Now, for concurrent executions of parallel components, the coupling overhead depends on the sequencing of components and their internal load balancing. Assuming a platform with a set Π of processors, in Figure 2.13 we illustrate two different configurations of a fully concurrent execution for (C_0, C_1) within the sequential CCS. Note that in similar figures throughout this section, white boxes are representative of the time spent in internal computation of the components while gray boxes correspond to their idle times. Finally, red boxes represent the time spent during coupling. In Figure 2.13a, the components run on processor sets $\mathcal{P}$ and $\mathcal{Q}$ respectively, such that $\Pi = \mathcal{P} \sqcup \mathcal{Q}$, while the coupler runs on one processor, as expected. In this configuration we assume that C_1 is not perfectly balanced and hence slower than C_0. In this case, C_0 waits for C_1 and the coupler can perform its work, or part of it, during this waiting time. The coupling overhead can therefore be totally or partially hidden by the internal imbalance of the slower component.

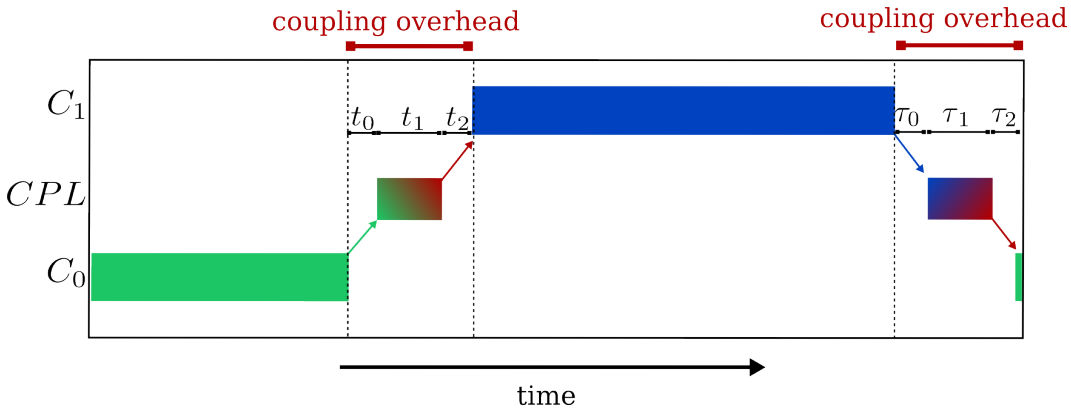
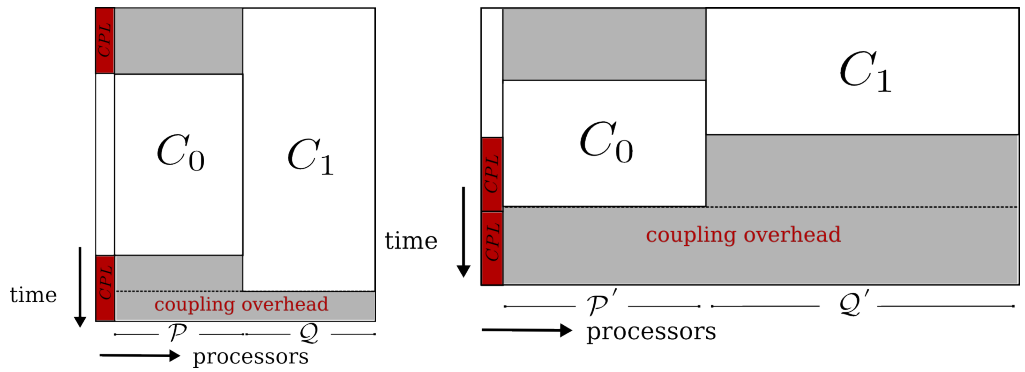


Figure 2.12: Coupling overhead within the CCS when components run sequentially in time. Example of one iteration.

However, for scalable components, it is expected that their elapsed run time will decrease with an increased number of processors. Therefore, for an actual coupled system, it is most likely that the time spent in the coupler becomes proportionally more important when the parallel efficiency of the components increases. In this case, the coupler may become a bottleneck for the simulation. Indeed, for high-resolution components running on a high number of processors, the time spent in the coupler becomes relatively more important when the elapsed time spent in the components decreases. An illustration of this problem is presented in Figure 2.13b where coupling overhead is important due to lack of load balancing at the coupling. In this configuration, we assume



(a) C_1 dominates elapsed time. (b) Coupler dominates elapsed time.

Figure 2.13: Two examples of the fully concurrent execution of components for the sequential CCS (where $\mathcal{P} \neq \mathcal{P}', \mathcal{Q} \neq \mathcal{Q}'$). In 2.13a the coupling overhead is partially hidden, as opposed to 2.13b.

that the components are scalable, thus increasing the number of processors to $\mathcal{P}'$ and $\mathcal{Q}'$ results in minimizing their elapsed time. As expected the coupling overhead is not minimized, since the coupler is sequential, and it is now less hidden.

A natural way to solve the above load balancing problem would be to employ a parallel coupler that follows the parallel CCS. In this model, communications are still centralized via a coupler but the latter is executed in parallel. Following, we examine the load balancing problem that appears in a parallel coupler, based on the CPL7 coupler of CCSM4 climate modeling. Under this context, CPL7 offers a flexible component layout across processors in an attempt to optimize both parallel efficiency and component throughput. Figure 2.14 presents some of the potential component layout available in CCSM4 for the classic climate model, running on a total of 128 processors ([22]) (again boxes are indicative of the elapsed time of one iteration of the coupled simulation). Here ICE stands for ice component, ATM for atmosphere, LD for land and OCN for ocean. Couplings occur between ATM-LD, ATM-ICE and ATM-OCN.

First, Figure 2.14a illustrates the fully concurrent layout of components (in both time and computational resources). In this layout, the set Π of processors is divided into independent subsets $(\mathcal{X}, \mathcal{Z}, \mathcal{R}, \mathcal{P}, \mathcal{Q})$. Each component is mapped onto one separate subset. For instance, the ATM component is mapped on subset $\mathcal{P}$, while OCN is mapped on subset $\mathcal{Q}$. In the parallel CCS, the coupler must also be mapped on processors, here CPL is mapped onto subset $\mathcal{X}$. On the opposite side of the spectrum, Figure 2.14b represents the fully sequential layout. Here, each component is distributed to all processors but components are executed sequentially with respect to each other.

In general, the effectiveness of running components concurrently is a trade-

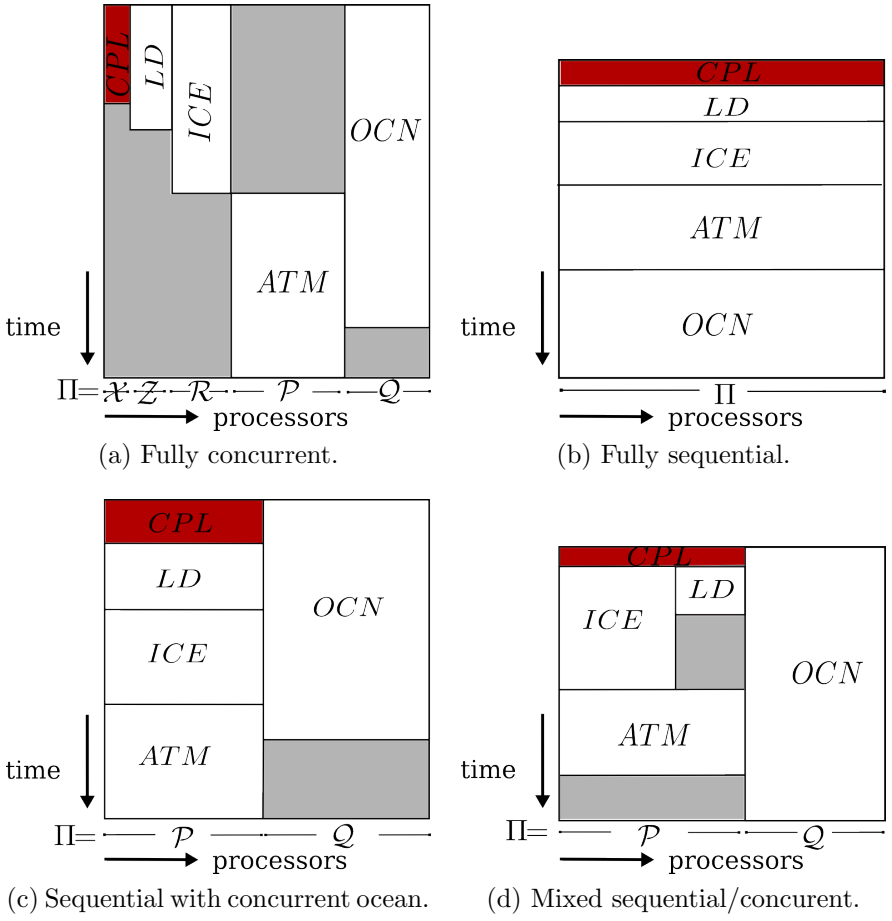


Figure 2.14: Four different processor layouts for the coupled simulation of earth's climate. Each figure is an example of one iteration for the coupled simulation.

off between the idle time created by concurrent execution versus the generally sub-linear scaling of components as processor counts are increased. Additionally, the optimal layout and processor count depend heavily on the application and the type of coupling between components. The scaling performance and spatial resolution of each component are also essential in determining an efficient layout. In this example, the fully concurrent layout does not optimize the idle time because of sequencing limitations (ATM can not start before ICE or LD) and the relatively small number of processors assigned to the whole coupled simulation (i.e. $\Pi = 128$). However, in general terms, the fully concurrent layout is better than the fully sequential one, since it has higher performance potentials as the number of processors increases. Indeed, this is not the case for the fully sequential layout that is limited by poor time and memory scalability of the components.

As a result, for this example, more efficient layouts include mixed sequential and concurrent solutions as depicted in Figures 2.14c and 2.14d. Those layouts combine the benefits of both sequential and concurrent executions. More precisely, through sequential execution, the idle time of components with sequencing limitations is minimized (e.g ATM and ICE), while the scalability of the coupled simulation is increased with the concurrent execution of other components (e.g. ATM and OCN).

Note that in this communication scheme (parallel CCS), the coupler is viewed as a separate component and it is attributed a number of processors. Under this context, the load balancing of the coupler should also be considered. Remember that the number of processors that are attributed to the coupler in CCSM4 is often determined based on the load balancing and optimization of the other components first. The above solution suggests that the components share with the coupler additional information about their internal partition and processor count that corresponds to the coupling data. As a result, the coupler maintains the same partition of the coupling interfaces placed on the same processor count for each component that participates in the coupling. For scalable components this may lead to a load imbalance of the coupled simulation during the coupling process. Additionally, within the coupler, the communication costs that correspond to data exchanges among different components are not optimized. This is not surprising since the data distribution of each component is performed independently from one another, thus the inter-component communications are not minimized during the partitioning procedure.

Finally, we discuss the load balancing problem, when the DCS is used, where components are mainly executed in fully concurrent layout as shown in Figure 2.15. This scheme is very promising since there is no explicit coupler and the components directly communicate with each other. An example of a coupling framework that uses the DCS scheme is OpenPALM. Here, coupling is performed exclusively by the processors of each component that participate

in the coupling process and own parts on the coupling interface. Again, as in the parallel CCS, each component uses its internal partition and processor subset when exchanging data with another component which may lead to load imbalance during the coupling process. Additionally, in the DCS, components exchange data directly on their mesh structures without the use of a coupler. Again, since the partition of the coupling data is done independently for each component, the inter-component communication may not be optimized.

To conclude, we notice that current coupling architectures try to minimize the imbalance of a coupled simulation by providing efficient processor layout and sequencing but do not address the load imbalance that appears during the coupling process. Indeed, even when the parallel CCS or the DCS are employed as communication schemes, the data distribution related to the coupling data is not well balanced. Moreover the inter-component communication may not be optimized. More precisely, when components are executed in the non-coupling mode (similarly to the standalone execution), their execution is governed by computational operations and their load is balanced due to an internal partitioning. However, when components enter a coupling process their operations change, due to data exchanges, and lead to a communication-intensive execution. This change on the type of operations between coupling and non-coupling modes is not reflected in the internal partition of each components and results in imbalanced couplings. In the next section, we explain in more details the above problem, providing illustrative examples.

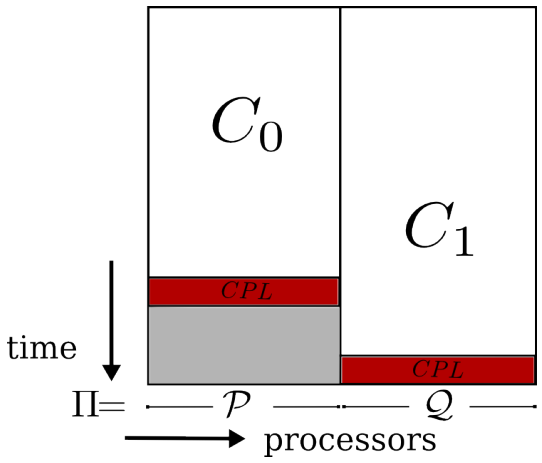


Figure 2.15: Fully concurrent execution of two components within the DSC.

2.4 Positioning

Nowadays, most coupled simulations use a straightforward approach to address the data distribution problem which simply treats each component as

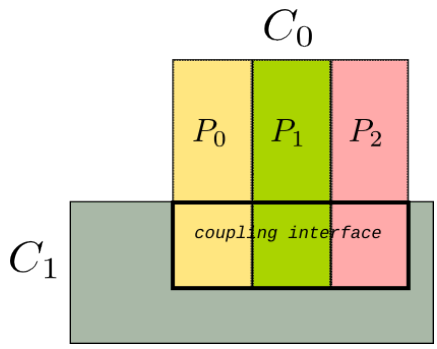
a completely independent simulation. As a result, the data decomposition is done independently and each computational domain is partitioned separately with a classic partitioning algorithm.

As explained before, the coupling framework determines how many processors will be assigned to each component, following a specific processor layout that optimizes the efficient execution among components (component throughput). Here, we assume a fully concurrent layout, where N components are mapped to distinct sets of processors. Therefore, one obtains N independent partitions, each of which divides the mesh of the corresponding component into the desired number of parts. As a final step, all parts from the N partitions are assigned to the processors of the parallel machine.

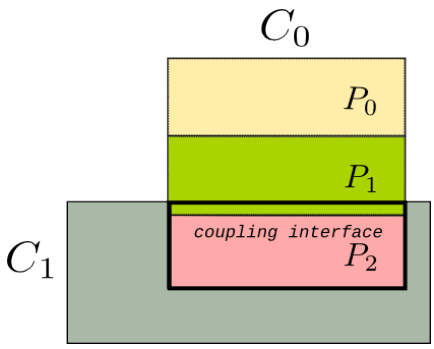
The above approach to partition coupled simulations, which we call *naive approach*, is the most commonly used method regarding the data distribution of a coupled simulation. Additionally, as far as we know, little work has been done to understand and address this problem in a more systematic way. While the naive approach is obviously easy to use and solves the partitioning problem for each component separately, it does not address the major challenges of coupled simulations, that is, the load imbalance during the coupling process and the inter-component communication costs. Indeed, for the non-coupling execution the internal load of each component is balanced and the internal communication costs are minimized. However, if we focus just on the coupling interface, we observe that the load might be highly imbalanced.

To illustrate the potential load imbalance, we consider in Figure 2.16 a coupled simulation with two components (C_0, C_1) and two different partitioning results obtained on the computational domain of C_0. C_1 is also depicted here, but its data distribution is not important for the purpose of this example. Note that both partitions in 2.16a and 2.16b equally balance the load of C_0 that corresponds to the entire computational domain (dividing it in three parts) and both minimize the internal communication costs (same edgecut). As a consequence, they are optimal from the point of view of the naive approach. However in Figure 2.16b, one may observe that the load related to the coupling interface is highly imbalanced as opposed to the load of the coupling interface in Figure 2.16a. More precisely, the latter (Figure 2.16a) is equally balanced both on the entire domain and the coupling interface. In Figure 2.16c, we illustrate the execution time and processor layout of the coupled simulation that corresponds to the data distribution of C_0, shown in Figure 2.16a. In a similar way, Figure 2.16d corresponds to Figure 2.16b. Note that for C_1 we assume a balanced load distribution during the coupling, so that we can focus on C_0. As one may see in Figure 2.16d, the time spent in the coupling process for C_0 may be significantly higher because of the load imbalance among the processors involved in the coupling. We also remark that two out of three processors of C_0 are (almost) idle during the coupling. Consequently, the above example shows in a simple way that the naive approach does not consider

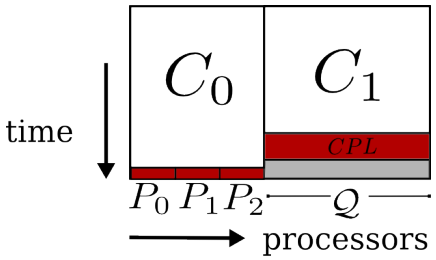
the coupling as a different phase and hence does not guarantee explicit load balancing throughout coupling.



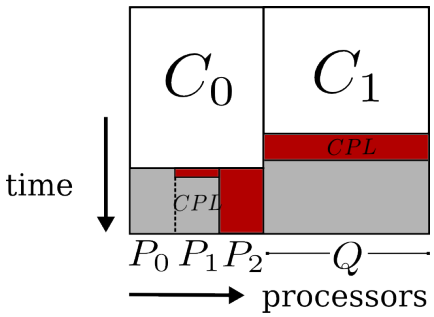
(a) Balanced partition on coupling.



(b) Imbalanced partition on coupling.



(c) Execution time of one iteration for the coupled simulation corresponding to Figure 2.16a.



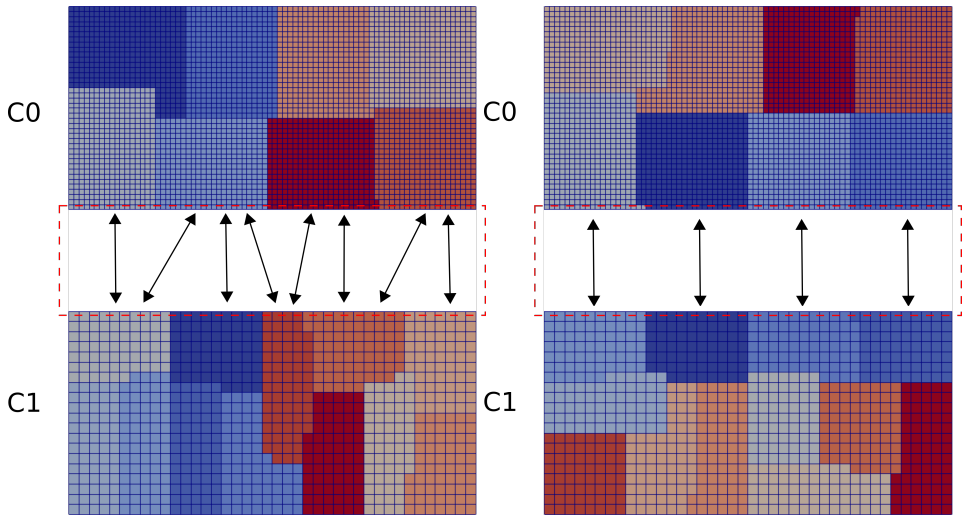
(d) Execution time of one iteration for the coupled simulation corresponding to Figure 2.16b.

Figure 2.16: Example of load balancing problem of coupled simulations.

In addition, within a coupled simulation the inter-component communication costs are often not optimized. This is not surprising if we consider that the data of different components have been partitioned completely independently from one another. Note that the total volume of inter-component communications remains rather constant since it corresponds to essential data exchanges that are imposed by the coupling dependencies between components. As a result, our objective here is to minimize the maximum volume of communication per processor and to minimize the number of messages exchanged among components.

Let us consider an example of a coupled simulation with two components (C_0, C_1) that exchange data on their coupling interfaces, as shown in 2.17. In Figure 2.17a, we show the data distribution on the computational domain of each component as it is computed by the naive approach. Let us assume that

the partition of both components is balanced in eight and twelve parts, respectively, and that the internal communication costs are minimized. However, we may see that the processors of C_0 that have data on the coupling interface (four processors here) are not "well-aligned" with the processors of C_1 that also have data on that interface (five processors). As a result, the number of messages that are exchanged between the two components is not optimal (8 inter-component messages). Ideally, if we use a partitioning algorithm that takes the coupling process into account, the number of inter-component messages may be reduced. In Figure 2.17b, one may see such a partitioning result, where the data of each component are distributed in such a way, that the number of inter-component messages is reduced to 4 with an optimized communication scheme.



(a) Naive approach. (b) Optimized approach.

Figure 2.17: Number of inter-component messages of a coupled simulation with two different partitioning strategies. The coupling surfaces for both components are noted with the red dashed square line (1D coupling).

As a conclusion, in this work, we address the problem of data distribution for coupled simulations where two (or more) components are combined together in order to solve highly complex physical systems. In this context, we observe that there are still challenges concerning the load balancing of the whole simulation.

In Chapter 4, we propose new graph partitioning algorithms that address this problem. Our objective is to efficiently decompose the data of a coupled simulation as part of an interactive system and not just as part of independent components. In this way, the load is balanced both during internal computations for each component and during couplings while the inter-component communications may be reduced. Additionally, with a partitioning method

that is aware of the coupling process, one may explicitly control the number of processors allocated in the coupling for each component. This offers an additional flexibility to the architecture of coupled simulations and is important for components with highly different load characteristics. We call the procedure of finding such data distributions the *co-partitioning*.

Moreover, in Chapter 3 we address the problem of graph partitioning with initially fixed vertices. We believe that an algorithm that supports such graph structures may improve the quality of our co-partitioning solutions. In both Chapters 3 and 4, we evaluate the results of our work with a series of experiments. Finally, in Chapter 5 we give our conclusions on this work along with some future prospectives.

Chapter 3

Graph Partitioning with Initial Fixed Vertices

3.1 Introduction

In this chapter, we study a variant of the classic graph partitioning problem, that is, the *graph partitioning problem with initially fixed vertices*. In general, this modified instance of graph partitioning typically appears when the underlying application imposes additional constraints on the assignment of computations to specific processors. For instance, an application may have an a priori knowledge on data locality for certain computations. This information can be modeled with the presence of fixed vertices in the initial graph and may guide the partitioning procedure into reaching solutions of better quality. In particular, a fixed vertex is a vertex whose partition assignment is predetermined as part of the input description and shall not be moved throughout the partitioning. Note that in the remainder of this chapter, we refer to the other vertices, that may move to any part, as *free*.

The work presented in this chapter is motivated by our interest in solving the load balancing problem of coupled simulations. More precisely, we believe that graph partitioning algorithms for problems with initially fixed vertices may be part of this solution. As explained in Chapter 2, classic graph partitioning strategies may fail to guarantee load balancing throughout the entire execution of coupled simulations. This is because different components do not just run as standalone simulations, but also interact with each other through dedicated coupling periods. Indeed, when components interact with each other and exchange data on their coupling interfaces, the operations involved in the process may not be well balanced. In this case more advanced partitioning algorithms should be considered. In this work, we propose load balancing solutions for such simulations and we introduce fixed vertices in order to influence the partitioning results between different components or different types of operations (computationally intensive or communication intensive). In other words,

the presence of fixed vertices may help the partitioning to successfully balance the load throughout the overall execution of such simulations and ultimately improve their makespan.

In the literature, other problems that use the fixed vertex paradigm are drawn from various areas, such as dynamic load balancing or integrated circuit design (VLSI). Here, we briefly present these examples. In scientific computing, a well-known example where the fixed vertex paradigm occurs is the load balancing of adaptive scientific computations [31]. In such applications, the discretization of the computational domain changes irregularly over time, leading to imbalanced load even for initially well-balanced simulations. The above feature gives rise to the repartitioning problem that addresses the difficulties of maintaining a dynamically changing load at runtime [18].

Note that the repartitioning problem is often modeled with graph or hypergraph structures and assumes an initially balanced partition. The additional requirement of repartitioning is to minimize the migration volume that appears when moving data among processors following a new partition. Under this context, a common approach to solve the repartitioning problem is to enrich the initial graph (or hypergraph) with one fixed vertex per part. Additional edges should be used in order to connect each fixed vertex to all free vertices of its respective part. These edges are called migration edges since they represent the cost of migrating data from the former partition to the new one. As a result, fixed vertices along with migration edges successfully model the additional constraint of repartitioning. Thus a good approach to solve the repartitioning problem is to perform a biased partitioning of the enriched graph, that accounts for minimizing the migration costs (in addition to minimizing the regular communication costs).

Graph partitioning with initial fixed vertices may be also used for non-numerical applications, such as the top-down placement technique [16] used in the integrated circuit design. Related studies [16, 27] demonstrate the importance of modeling the above problem with fixed vertices, proposing new partitioning approaches that account for extra constraints. The authors suggest that the presence of fixed vertices represents more accurately any positioning information, such as the locations of external pin connections, or the potential locations of certain cells in the final placement. In this context, the number of fixed vertices can reach up to 50% of the total number of vertices.

The main contribution of this chapter is a new algorithm, named KGGGP, that finds a direct k-way graph partitioning, extending a classic greedy approach for bipartitioning. Though KGGGP addresses the general problem of graph partitioning, its best results appear mainly when fixed vertices are used during the partitioning procedure. To complete our study we present a systematic comparison of different partitioning algorithms that address the above problem. What we observe is that for the graph partitioning problem with initial fixed vertices, KGGGP exhibits better partitioning results compared to

state-of-the-art partitioning tools, which use Recursive Bisection (RB) based techniques. This is an interesting result that motivates us to understand what happens when RB is used to partition a graph with fixed vertices.

3.2 Issues of Recursive Bisection

The motivation behind our study comes from the observation that RB based algorithms produce partitions of lower quality when the fixed vertex paradigm is involved, a remark initially shared in [6]. Here, we attempt to further explain this behavior.

More precisely, RB based methods follow a divide & conquer strategy: the original graph is first split in two parts (bisection) and this procedure is recursively repeated independently on the two resulting subgraphs, until the desired number of parts is obtained. Thus, it is possible to represent this procedure with a bisection tree (Figure 3.1) which illustrates the implicit part numbering scheme used by RB.

Let us now assume that the number of desired parts at the end of the partitioning is k. Note that at each bisection step, RB selects half of the available part numbers and assigns them to one of the two resulting subgraphs. This step has a complexity of $O(k!)$ since there are combinatorially many part numbering selections for each bisection step. Therefore, in order to avoid choosing among $k!$ combinations, RB decides to blindly group together parts with consecutive part indices. For instance in the first bisection, RB will try to assign the first half of all the part indices, $[1, k/2]$, to one subgraph and the second half, $[k/2 + 1, k]$, to the other. At each level of the bisection tree, the same methodology is applied.

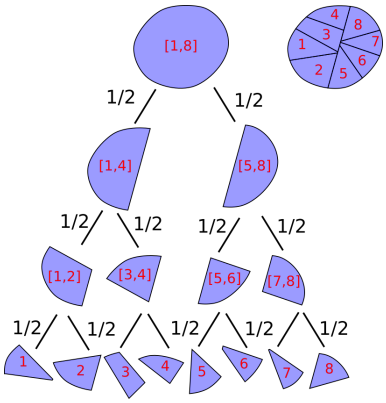
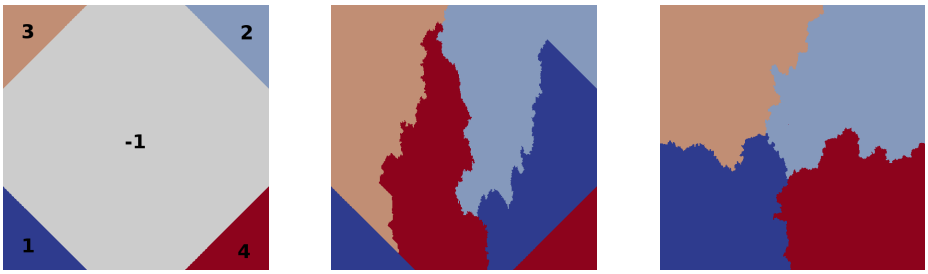


Figure 3.1: Illustration of the RB method: bisection tree and its implicit part numbering (in red) for a 8-way partition.

In Figure 3.2, we illustrate a simple but compelling example that exhibits the partitioning issues emerging when a RB algorithm partitions a graph with

fixed vertices. More precisely, as one may see in 3.2a, we use a grid graph (of dimensions 1000×1000) with an initial part numbering of fixed vertices such that vertices near the corners are assigned accordingly to 4 different parts. We consider the rest of the vertices as free (part -1). Following, we partition the whole graph in 4 parts. We compare two different methods implemented within the same multilevel framework and tuned with the same parameters. To make the analysis easier, the refinement algorithms have been disabled. In Figure 3.2b, we present the result obtained by the RB method (implemented by SCOTCH), while in 3.2c, we present the same result obtained by our KGGGP method, which will be described later in this chapter. Here, one may clearly see that during the first recursion level of RB it is not possible to select a good bisection between parts $[1, 2]$ and $[3, 4]$ which respects the constraint of initial fixed vertices. Thus, the final partition quality is considerably poor for the RB based method. Indeed, when the part numbering of fixed vertices conflicts with the inherent numbering constraint of RB, the method can not successfully respect both constraints, leading to largely disjoint parts and bad edgecut.



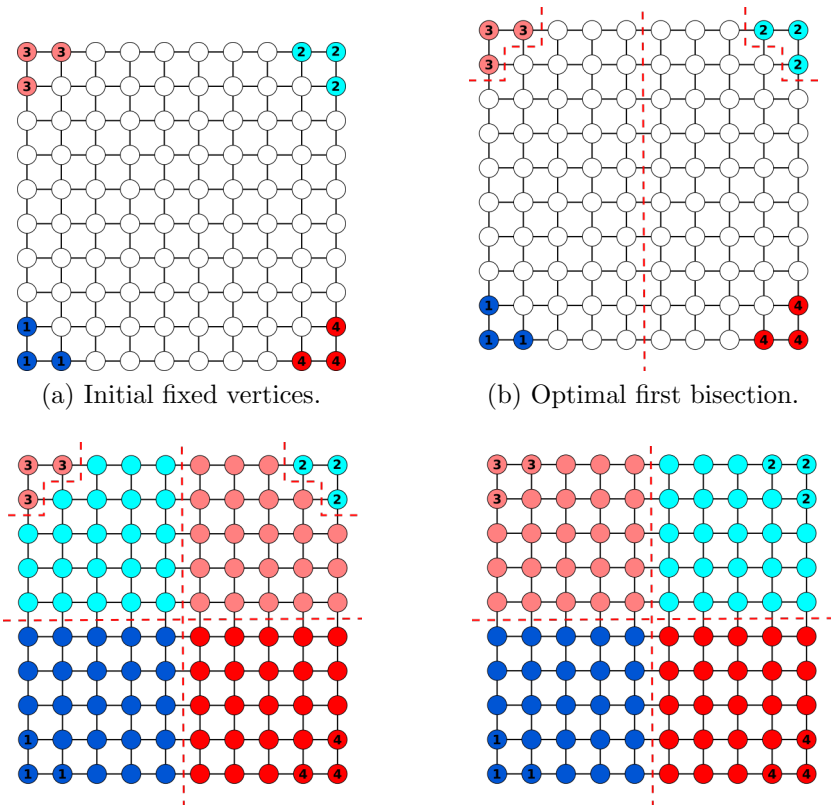
(a) Initial fixed vertices. (b) RB. (c) KGGGP.

Figure 3.2: Given initial fixed vertices, comparison of two different partitioning methods (RB and KGGGP). Results of a 4-way partition of a 1000×1000 grid graph: the RB method fails to extend an initial partition while the KGGGP method succeeds.

To better understand what goes wrong in this case, let us consider the Figure 3.3 that reproduces the same experiment on a smaller 10×10 grid graph. Here, we realize the same 4-way partitioning under the constraint of fixed vertices depicted in Figure 3.3a. Following a similar methodology as *Simon and Teng* [63], one may clearly see that RB does not succeed in finding the optimal solution under the constraint of fixed vertices. This is because RB tries at each step to find the optimal local solution, and in this case the first optimal bisection involves disconnected components as shown in Figure 3.3b (i.e. vertices fixed to 3 are disconnected from the part containing vertices fixed to 4 and the same happens for vertices fixed to parts 2 and 1). Note that RB cannot find a better bisection that puts fixed vertices of parts 1 and 2 in a single connected component. Another possible solution for the first bisection would give edgecut of higher cost (not optimal). Therefore, the first "bad" bisection

of RB is maintained in the final solution, which is clearly not optimal: the best solution of RB is presented in Figure 3.3c compared to the optimal solution shown in Figure 3.3d.

Note that the above examples are just an illustration of the problematic behavior of RB methods with initial fixed vertices. Further experiments that confirm our observations follow in Section 3.5.



(a) Initial fixed vertices.

(b) Optimal first bisection.

(c) Best solution for RB.

(d) Optimal solution.

Figure 3.3: Issues of the RB method under the constraint of initial fixed vertices. While finding the optimal first bisection, the best solution of RB to the 4-way partitioning problem will not be optimal, showing an additional cost of 8 edges cut.

3.2.1 Graph Partitioning Algorithms with Fixed Vertices

In Table 3.1, we present some useful information about common graph and hypergraph partitioning tools, such as Scotch, Metis or PaToH. First, we note that most partitioning tools use the widely adopted RB heuristic combined with the multilevel framework (MLRB or MLKW with RB for the

initial partitioning phase). The above observation is not surprising, since these methods have been proven to be very efficient for the classic graph partitioning problem and are viewed as the state-of-the-art. Another interesting remark is that, despite the research interest on the partitioning problem with initial fixed vertices, more than half of the given tools do not handle at all this specific problem. Note that this is especially true for graph partitioning tools. More importantly, after testing some of the tools which actually provide solutions for graph/hypergraph partitioning with fixed vertices, we observed that they do not successfully minimize the edgecut, resulting most of the times in lower quality partitions. Following the explanation in Section 3.2, we credit the above behavior to the use of the RB heuristic, which provides great results for the classic graph partitioning problem but may fail to properly handle the presence of initial fixed vertices.

Nevertheless, there are some interesting studies about partitioning algorithms that successfully handle initial fixed vertices and we review them here.

An algorithm that addresses graph problems which involve initially fixed vertices is the one proposed in RM-Metis, where the adaptive object space decomposition is modeled as a graph instance [5]. However, the proposed algorithm addresses only the k-way repartitioning problem and requires an existing partition of a graph that has become imbalanced. The main idea of the algorithm is to apply greedy growing techniques, selecting $k-1$ growing parts and a shrinking one, until the load of every part drops below the maximum allowed part size. As mentioned before, in such re-balancing problems fixed vertices help the repartitioning procedure to re-distribute the load more efficiently. Note that RM-Metis has limitations to the number of initial fixed vertices that may be used during the repartitioning, namely one per part, whereas kGGGP has no such restrictions. Finally this algorithm is implemented inside the multilevel framework of Metis but unfortunately, as far as we know, its implementation is not publicly available.

Following we describe in detail the two alternative methods that solve the partitioning problem with initial fixed vertices. In the next section, we review the RBBGM algorithm that was first introduced in kPaToH and then, we review the kGGGP method in Section 3.4.

3.3 Recursive Bisection with Bipartite Graph Matching (RBBGM)

An algorithm for hypergraph partitioning is introduced in [6] that successfully handles problems with initially fixed vertices. In this work, the authors identify the inferior performance of RB when fixed vertices are involved in the process, mentioning its inability to explore the combinatorially many part labelings that correspond to a given fixed vertex configuration. To correct the above

Table 3.1: Graph and hypergraph partitioning tools.

Tools	Type	Fixed	Parallel	Scheme	Initial Part.	Available
METIS [40]	graph	no	no	MLRB	–	source
KMETIS [39]	graph	no	no	MLKW	RB	source
ParMetis [40]	graph	no	yes	MLKW	RB	source
Scotch [53]	graph	yes	no	MLKW	RB	source
PT-Scotch [53]	graph	no	yes	MLRB	–	source
RM-Metis [5]	graph	only k	no	MLKW	greedy	no
KaFFPa [59]	graph	no	no	MLKW	RB	source
Chaco [48]	graph	no	no	MLRB	spectral	source
HMetis [40]	hypergraph	yes	no	MLRB	–	binary
KHMetis [40]	hypergraph	no	no	MLKW	RB	binary
PATOH [77]	hypergraph	yes	no	MLRB	–	binary
KPATOH [6]	hypergraph	yes	no	MLKW	RBBGM	no
ZOLTAN (PHG) [1]	hypergraph	yes	yes	MLRB	–	source
Mondriaan [68]	hypergraph	no	no	MLRB	–	source

deficiency, they propose a new multilevel direct k-way hypergraph partitioning that uses a RB-based algorithm and an additional post-processing technique to relabel the resulting parts such that the edgecut remains minimized. They refer to the above algorithm as RBBGM and they provide an implementation called KPATOH based on modifications of the multilevel framework of PATOH.

For the coarsening phase of KPATOH, a modified Heavy Connectivity Matching [1] is proposed such that no two fixed vertices are matched together at any coarsening level. However, a fixed vertex can be matched with any free vertex, forming a fixed super-vertex for the next level. Therefore, the number of fixed super-vertices in the coarsest level is equal to the number of initially fixed vertices of the hypergraph. As always, free vertices are matched together according to the chosen heuristic.

During the initial partitioning phase of KPATOH, fixed vertices are temporarily removed from the coarsest hypergraph and a partitioning of the resulting hypergraph is performed with a classic RB-based algorithm. Once the partition is computed, fixed vertices are re-introduced in the hypergraph according to a relabeling strategy. Note that if no relabeling is used, fixed vertices are simply re-assigned to the parts based on their initial part numbering. In this case, the final partition may not be optimized in terms of net cost minimization, since nets incident to fixed vertices are not considered during RB. In other words, a relabeling strategy that minimizes the net cost contribution of re-introduced fixed vertices is necessary to obtain an optimized partition for the coarsest hypergraph.

The problem of relabeling fixed vertices is formulated in KPATOH as a maximum weighted bipartite graph matching problem, that represents the minimum increase of edgecut. In the proposed formulation, sets of fixed vertices and resulting parts form the two node sets of the bipartite graph $B = (X, Y)$. More precisely, each vertex X_i in B represents fixed vertices, initially assigned

[1] HCM is the equivalent algorithm of HEM for hypergraph partitioning

to part i, while each vertex Y_i represents vertices that belong to part i after the partitioning. Additionally, the bipartite graph contains all possible edges (X_i, Y_j) between fixed vertices and ordinary ones with a weight that corresponds to the sum of weights for edges with incident vertices in both X_i and Y_j.

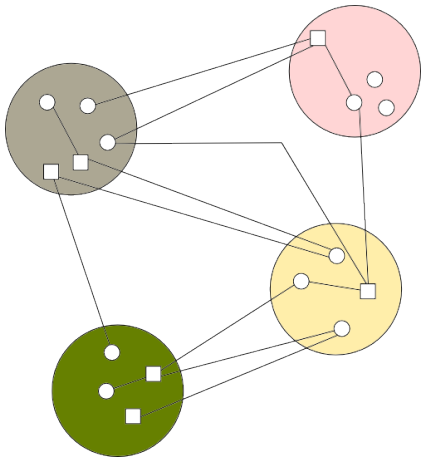


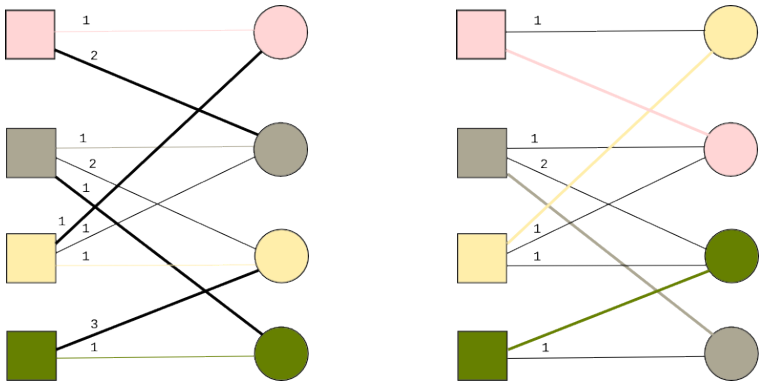
Figure 3.4: Example of coarsest graph where each color corresponds to a different part. Fixed vertices are represented as squares and free ones as circles.

Following, we give an example of the initial partition phase of KPaToH on a graph structure rather than on a hypergraph for reasons of consistency with the focus of this work. Figure 3.4 shows the partitioning result of a graph with fixed vertices using KPaToH and no relabeling strategy is used to reassign fixed vertices. That is, the modified graph which contains only free vertices has been partitioned in four parts with RB while fixed vertices have been simply re-introduced in the graph after the partitioning, maintaining their initial part assignment. Note that fixed vertices are represented in the figure as squares while free vertices as circles. Additionally, for ease of presentation, unit edge weights are assumed and only edges connecting fixed vertices and free ones are displayed, since any additional edgecut contribution may be due to such edges.

In this partitioning example, the upper bound of edgecut contribution after fixed vertices are re-introduced in the graph is 14 (edges). In Figure 3.5, we present two instances of the bipartite graph that models the re-assignment of fixed vertices for this example, one (3.5a) that corresponds to no relabeling and a second one (3.5b) which follows a relabeling strategy. Finally, note that sets of fixed and (previously) free vertices that are assigned to the same part are drawn in both figures with the same color.

Therefore, in the case of no relabeling, the edgecut increase is 10 and implies an edgecut saving of four (colored edges). However, it is easy to see that there is an assignment of fixed vertices that leads to higher egdecut saving and is

the solution of the maximum-weight bipartite graph, illustrated in 3.5b. This relabeling obtains the highest saving of edgecut which leads to the minimum cost increase of 14-7 = 7, instead of 10.



(a) No relabeling.

(b) Relabeling strategy of minimized cost contribution.

Figure 3.5: The bipartite graph used for the reassignment of fixed vertices in the example 3.4. A square vertex represents a set of fixed vertices X_i, while a circle vertex represents a part i

Finally, for the uncoarsening phase, a modified version of the k-FM refinement algorithm is used where fixed vertices are locked to their respective parts and are not allowed to move between parts.

Experiments on hypergraphs with fixed vertices performed by KPATOH show a net cost improvement of overall average between 17% and 21% compared to the multilevel RB-based method used in PATOH. Unfortunately, as it is mentioned in Table 3.1 KPATOH is not available; therefore we decide to implement our own version of RBBGM for graph structures described below.

Discussion on Graph Implementation of RBBGM

We choose to implement RBBGM inside the multilevel framework of SCOTCH for two main reasons. First, SCOTCH is one of the most widely used graph partitioning tools, with a fast implementation of the multilevel framework and an easy programming interface. But more importantly, SCOTCH already includes a RB-based method that addresses the problem of graph partitioning with initially fixed vertices that can be used as a reference method. Remember that the multilevel framework consists of multiple heuristics and each partitioning tool has different parameters for the coarsening, initial partitioning and uncoarsening phase, that may heavily influence the quality or execution time of the final solution. Therefore, in order to conduct a reliable comparison of different methods used for the initial partitioning phase (such as RBBGM and

KGGGP), one shall implement them all inside the same multilevel framework. Finally, note that in order to solve the maximum weighted bipartite graph matching problem that appears in RBBGM, we use an implementation of the Hungurian algorithm [45] which finds an exact solution and has a complexity of $O(k^3)$.

3.4 The KGGGP Algorithm

In this section, we describe a direct k-way graph partitioning algorithm, called KGGGP (k-way greedy graph growing partitioning), which can be easily integrated in a multilevel framework and that successfully handles any number of initially fixed vertices.

To begin with, we briefly describe here the standard greedy approach for bipartitioning [21, 8] that has served as key idea for many partitioning algorithms and particularly for KGGGP. This greedy approach starts by placing a random *seed* vertex into each of the two parts and then the remaining vertices are iteratively added into the parts according to a minimization criterion.

3.4.1 Algorithmic description of KGGGP

The KGGGP algorithm is an extension of the standard greedy bipartitioning algorithm for a k-way graph partitioning, where a partitioning of k parts (instead of just two) is directly computed. In a certain way, KGGGP can be seen as a variation of the FM algorithm [29] with k growing parts and a *free* part, denoted as -1, which initially contains all free vertices and becomes empty at the end of the procedure. A detailed description of KGGGP is given in Algorithm 1 that will be discussed below.

As we mentioned above, greedy algorithms often use seeds to initiate the partitioning procedure, usually based on BFS (breadth first search) [25]; however in KGGGP, the use of seeds is optional. As an alternative, the selected minimization criterion determines the first vertex displacement for each part.

At each step of the main loop, the KGGGP algorithm selects the best *global* displacement (v, p), among all free vertices and all possible parts. Note that we initially consider all free vertices in part -1 as candidates to move to any of the k parts and we choose the best displacement, based on an edgecut minimization criterion (gain), subject to the balance constraint. Additionally to the above criteria, the algorithm enforces the selection of a displacement (v, p), such that v is connected with vertices already assigned to the part p (connectivity constraint). In other words, one prefers to select a vertex v in the neighborhood of the growing part p, as far as possible.

In order to quickly locate the best displacement, we use a similar data structure as in FM adapted here for k parts. This structure, called *gain bucket*

data structure, maintains a sorted list of displacement gains. To implement the bucket, we use an array whose i^{th} entry contains a doubly-linked list of all displacements with gain currently equal to $g = max\_gain - i$ (Figure 3.6). Additionally, an array containing references of all displacements is used to perform quick gain updates in the same way as in FM. More precisely, in KGGGP, a two dimensional array is employed which can be accessed by vertex and part numbering (v, p) allowing the updates of neighbor vertices after a displacement in constant time.

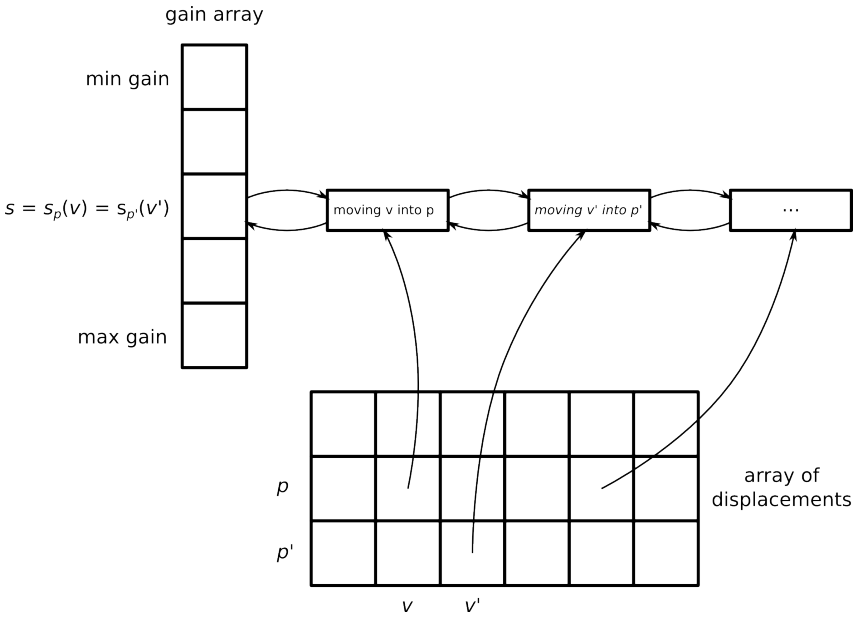
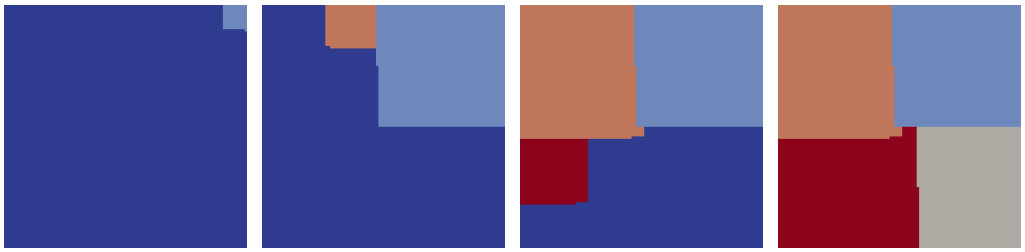


Figure 3.6: Bucket data structure (inspired from FM algorithm), illustrating two displacements (v,p) and (v',p') having the same gain g that is currently the best possible gain.

The KGGGP algorithm uses three instances of the gain bucket structure to store and select displacements: H_{REG} initially contains all possible *regular* displacements (line 2), while H_{NCC} and H_{NBC} store displacements that do not respect the connectivity and balance constraint respectively. Note that at first, H_{NCC} and H_{NBC} are both empty (lines 3–4). In the main loop of the algorithm, the best displacement is selected initially from H_{REG} and if it respects all constraints, the chosen displacement will be applied. However, each time we encounter a displacement that violates the connectivity or the balance constraint, we move it to the appropriate bucket (H_{NCC} and H_{NBC} respectively). It is expected that at a given time, bucket H_{REG} will become empty while buckets H_{NCC} and H_{NBC} will contain the rest of possible displacements. In this case, until the algorithm terminates, we repeat the above procedure searching for displacements in H_{NCC} and if they do not respect the balance constraint, we move them into bucket H_{NBC}. Finally, when H_{NCC} becomes

empty, the only displacements left for selection are included in H_{NBC}.

Once a displacement (v, p) is chosen (line 29), v is moved to the corresponding part p (line 31) and then (v, p) is removed from the respective bucket (line 33). Additionally, we remove from any bucket all possible displacements of the same vertex to other parts (line 34) and we update the selection criterion of its neighbors as in the FM algorithm (line 37). Finally, each time a vertex v is moved to a part p, the algorithm checks if displacements (v', p) in the neighborhood of v that previously violate the connectivity constraint, respect it once again. In this case, such displacements become "regular" and are moved from H_{NCC} back to H_{REG} (line 42), as explained in Figure 3.8.



(a) Step 100. (b) Step 3000. (c) Step 6000. (d) Step 10000.

Figure 3.7: Steps of KGGGP while partitioning a 100×100 grid graph in 4 parts. The free part is colored in blue. This part becomes empty at the final step (10000) showing a 4-way partition.

In Figure 3.7, we illustrate the evolution of the part growing when a simple 100×100 grid graph is partitioned into 4 parts using the KGGGP algorithm without a multilevel framework. As one may see, the algorithm aims to respect both the balance and the connectivity constraints leading to a rather balanced and connected partition even with no refinements.

3.4.2 Fixed Vertex Management

Note that it is very straightforward to handle initial fixed vertices within KGGGP. More precisely, fixed vertices are directly placed in their respective parts before the algorithm starts and are simply not considered as candidates for any displacement. Obviously, these vertices are not ignored: they influence the balance among parts and are taken into account in the gain calculation of free vertices in the neighborhood.

3.4.3 KGGGP in a Multilevel Framework

The KGGGP algorithm can be easily integrated in any multilevel framework with some simple adjustments regarding the initial fixed vertices. Firstly, during the coarsening phase, an extra constraint is added, so that fixed vertices

Algorithm 1 The KGGGP algorithm.

Input: graph $G = (V, E)$
Input/Output: partition array $part[]$ (of size $|V|$) initialized with fixed and free vertices

```
% initialization step of gain bucket structures (H_REG, H_NCC and H_NBC)
H_REG ← initialize with all displacements (v, p) of any free vertices v to any parts p
H_NCC ← ∅
H_NBC ← ∅
% main loop
while there are free vertices do
    % select the best displacement
    repeat
        if H_REG is not empty then
            (v, p) ← consider a displacement with maximum gain from H_REG
            if balance constraint is not respected for displacement (v, p) then
                move (v, p) from H_REG to H_NBC
            else if connectivity constraint is not respected for displacement (v, p) then
                move (v, p) from H_REG to H_NCC
            else
                choose (v, p)
            end if
        else if H_NCC is not empty then
            (v, p) ← consider a displacement with maximum gain from H_NCC
            if balance constraint is not respected for displacement (v, p) then
                move (v, p) from H_NCC to H_NBC
            else
                choose (v, p)
            end if
        else if H_NBC is not empty then
            (v, p) ← consider a displacement with maximum gain from H_NBC
            choose (v, p)
        end if
    until a displacement (v, p) is chosen
    % perform the chosen displacement (v, p)
    part[v] ← p
    % update buckets
    remove (v, p) from gain bucket structures
    remove (v, p') where p' ≠ p from gain bucket structures
    for all vertex v' adjacent to v do
        for all parts p' do
            update the gain of displacement (v', p') in gain bucket structures
            if (v', p') ∈ H_NCC and p' = p then
                move (v', p') from H_NCC to H_REG
            end if
        end for
    end for
end while
```

which belong to different parts can not be matched together, while they may be matched with free vertices. Following, we partition the coarsest graph with KGGGP as it is described above and we continue with the uncoarsening phase, where refinements for k-way partitioning are performed to further improve the final result. Note that during this phase, we maintain all fixed vertices locked, forcing them to remain in place.

3.4.4 Gain Formulas for Minimization Criterion

As we mentioned above, there exist multiple minimization criteria to determine displacement selection. Here, we present three of them: the *classic* gain minimization as it is presented in FM algorithm, the *diff* gain proposed by Battiti and Bertossi [8] and finally a *hybrid* minimization criterion.

Assuming that a vertex v moves into part p, we divide its incident edges (v, v') in three categories: the *internal edges* such that $part[v'] = p$, the *external edges* such that $part[v'] \neq p$, and the *free edges* such that $part[v'] = -1$. Let $N_{int}(v)$ be the number of internal edges, $N_{ext}(v)$ the number of external edges and $N_{free}(v)$ the number of free edges, for v. Clearly, $N_{int}(v)$ measures how strongly v is connected to the part p, while $N_{ext}(v)$ measures how strongly it is attracted to other parts (except -1). The parameter α is an integer constant that is used to enforce the part connectivity once again; more precisely, it favors internal edges compared to other edges. Typical values of α are in the range 1 to 10.

Table 3.2: Description of minimization criteria.

`Criterion`	`Gain Formula`
classic	$G = \alpha.N_{int}(v) - N_{free}(v)$
diff	$G = \alpha.N_{int}(v) - N_{ext}(v)$
hybrid	$G = \alpha.N_{int}(v) - N_{ext}(v) - N_{free}(v)$

Based on the above definitions, the Table 3.2 presents the gain formulas of the different criteria. The main difference between those formulas is due to the *free* edges, depending on whether they are considered as external or not.

3.4.5 Time and Space Complexity

The main steps of the KGGGP algorithm consist of initializing displacements, selecting displacements and updating the bucket structures after a displacement is performed. The initialization step needs to compute the gain for all possible displacements (v, p) of any free vertices v to any parts p. As the gain calculation depends on the neighborhood of each vertex v, the time complexity is $O(k|E|)$. Then, during the main loop, we select at each iteration the

"best" displacement using the three bucket structures. In Figure 3.8, we illustrate a diagram of all possible moves of a displacement between the three buckets during the selection phase. Each transition may happen exactly once for each displacement. In the worst case, a displacement (v, p) may be considered up to three times before being selected. This is the case when (v, p) is initially in H_{REG} but does not respect the connectivity constraint, so it moves to bucket H_{NCC}. However, due to a possible displacement of a neighbor vertex to p, vertex v becomes connected to the part and (v, p) moves back to bucket H_{REG} (lines 38–39). Note that this is an update step (dashed arrow) and is not considered in the complexity of the selection phase. Moreover, (v, p) may move to H_{NBC} if the balance constraint is not respected, before being finally selected as the best displacement. As a consequence, the time complexity of the selection phase is equal to $O(|V|)$. Note that the actual displacement of a vertex has constant time complexity. Furthermore, during the update phase, displacements that correspond to already assigned vertices should be removed from all buckets and then the gain of neighbors should be updated. That is, for each neighbor of a newly assigned vertex, it is necessary to update its gains regarding all k parts, leading to a complexity of $O(k|E|)$ for this step.

Therefore, the total complexity of KGGGP is $O(k|E|)$, considering that $|V|$ is dominated by $|E|$. As a reminder, the time complexity of RB is $O(log(k)|E|)$. As concerns the space complexity, it is mainly due to storing all possible displacements in the gain bucket data structure, which is $O(k|V|)$.

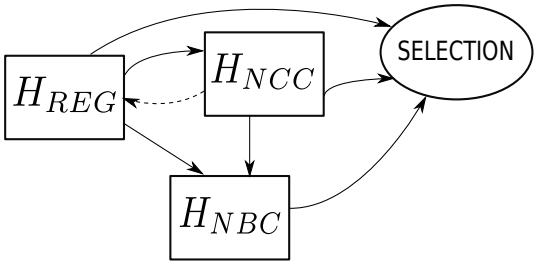


Figure 3.8: Diagram of all possible moves of a displacement during the selection phase.

3.4.6 Optimization: Local Greedy Approach

In order to reduce the total time complexity of KGGGP, we implement a second version of the method, where we enforce *local* selection of displacements instead of the *global* selection described above. The key idea here is to search for upcoming displacements only in the neighborhood of vertices that belong already to a part. This approach is similar to the one used in KMETIS to optimize the k-way FM refinement heuristic [39]. As a result, we do not need to initialize the H_{REG} bucket structure by computing *a priori* the gain value

for all possible displacements. Instead, after a displacement (v, k) is chosen, we dynamically insert in H_{REG} new displacements (v', k) for all neighboring vertices v' of v, that remain free. If H_{REG} becomes empty while the partition is not complete, the method switches back to the global approach for the remaining free vertices, giving a time complexity of $O(k|E|)$ in the worst case and $O(|E|)$ in the best case.

3.5 Experiments

In this section, we present the experimental results on the partitioning quality and time performance of different algorithms that solve the graph partitioning problem with initially fixed vertices. Our main goal is to analyze the performance of KGGGP and compare it to existing solutions.

For our experiments, we implement two versions of the KGGGP algorithm, one that follows the global greedy approach (KGGGP_G) and one that follows the local approach (KGGGP_L) as described in Section 3.4. Following, we compare the two versions of KGGGP with our RBBGM implementation of KPATOH (explained in 3.4) and the default RB-based method of SCOTCH. Both codes of KGGGP and the one of RBBGM are publicly available in the *MetaPart* library at http://metapart.gforge.inria.fr.

Note that we implement the above methods inside the same multilevel framework (that of SCOTCH), as part of the initial partitioning phase of a MLKW algorithm. Moreover, all methods are tuned with exactly the same partitioning parameters: imbalance factor of 5%, HEM for coarsening, maximum coarsest graph size equal to $30 \times k$, FM refinement with 10 passes and a maximum number of negative moves allowed set to 100 for each refinement pass. As a result, the above configuration allows us to fairly compare the impact of RBBGM, KGGGP and RB algorithm on the final partitioning solution. In the remainder of this section, we will refer to the default RB method of SCOTCH as "SCOTCH", and it will serve as a reference for the relative comparisons in the following experiments.

To perform the experiments, we use graphs that come from different scientific domains (numerical simulations, clustering problems and road networking) available in the public *DIMACS'10* collection [7], for experimentation on graph partitioning and graph clustering. One may find in Table 3.4 the different graph categories that are used in our experiments along with some useful information, such as the total number of vertices or edges, for each graph.

For the sake of brevity and readability of our experimental results, we include here two different types of figure. In Figures 3.11, 3.14 and 3.16 for each graph category we present normalized results relative to SCOTCH on the average values over all included graphs (within that category). Besides, each experiment is performed 5 times for every graph. In these figures, the x-axis

represents the edgecut and the y-axis represents the execution time, while the number of desired parts increases from 10 to 500. Moreover, to analyze our experiments in a global perspective, we include figures that demonstrate average values of the obtained results (either the edgecut or the execution time) over the entire graph collection or over significant groups of categories. Note that the error bars in those latter figures indicate the standard deviation for each method. Finally, whenever a method fails to compute a valid partition, its results are not taken into account in the average calculations. Obviously by doing so, we may favor methods that fail to respect the balance constraint. To address this problem, we carefully examine the success rate of each partitioning tool in addition to the main metrics (edgecut and execution time).

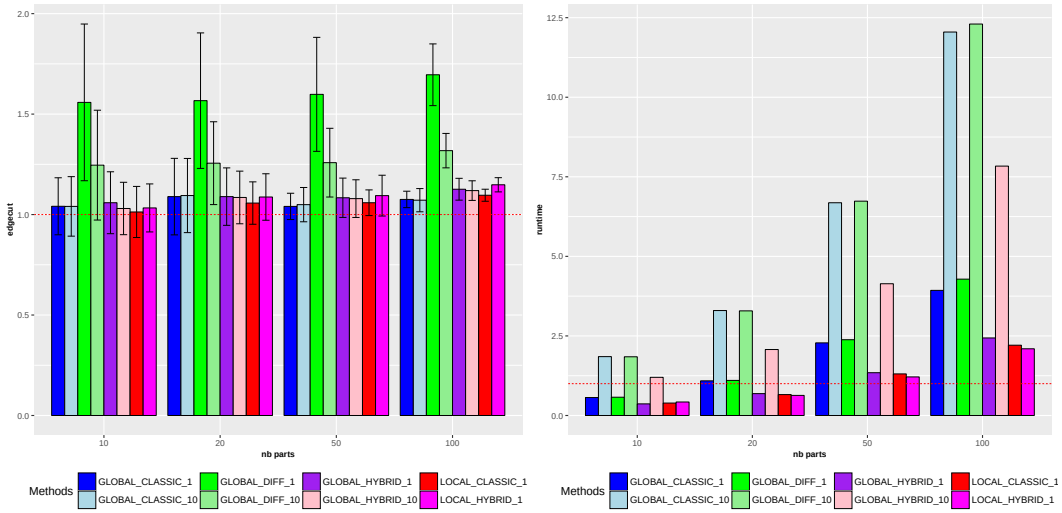
In this study, we perform three different experiments. The first one aims to evaluate the behavior of KGGGP algorithms for the classic graph partitioning problem, while the second one evaluates KGGGP and other existing algorithms for the graph partitioning with initially fixed vertices. Finally, since most tools that support fixed vertices are designed for hypergraphs, we include a last experiment that compares KGGGP with hypergraph tools.

3.5.1 Tuning of KGGGP without Multilevel Framework

The goal of this preliminary experiment is to tune some important parameters of the KGGGP method. More precisely, we want to evaluate the best gain formula (*classic, diff or hybrid*) and the impact of the *local* optimization (KGGGP_L) vs the *global* approach (KGGGP_G) on both quality and performance. Those parameters have been previously described in Sections 3.4.4 and 3.4.6.

To enable an easier analysis, we disable the multilevel framework for every method and we perform the experiment only on the Walshaw collection, that contains graphs of smaller size. The results are presented in Figure 3.9 relative to SCOTCH. Here we see that the *local* approach, tuned with the *classic* or hybrid gain formulas and $\alpha = 1$, provides the best results for both edgecut and execution time. More precisely, for KGGGP_L the *classic* formula gives a slightly better edgecut than the *hybrid*, while the *hybrid* one slightly improves the runtime performance.

Besides as concerns the performance results, one may see that the *local* approach KGGGP_L is around two times faster than the *global* one for almost the same edgecut, while the memory footprint (not presented here) is considerably reduced. As a conclusion, in the following experiments, we will use the *classic* gain formula with $\alpha = 1$ for both *global* and *local* methods, and we expect that KGGGP_L outperforms KGGGP_G for runtime performance.



(a) Edgecut.

(b) Runtime.

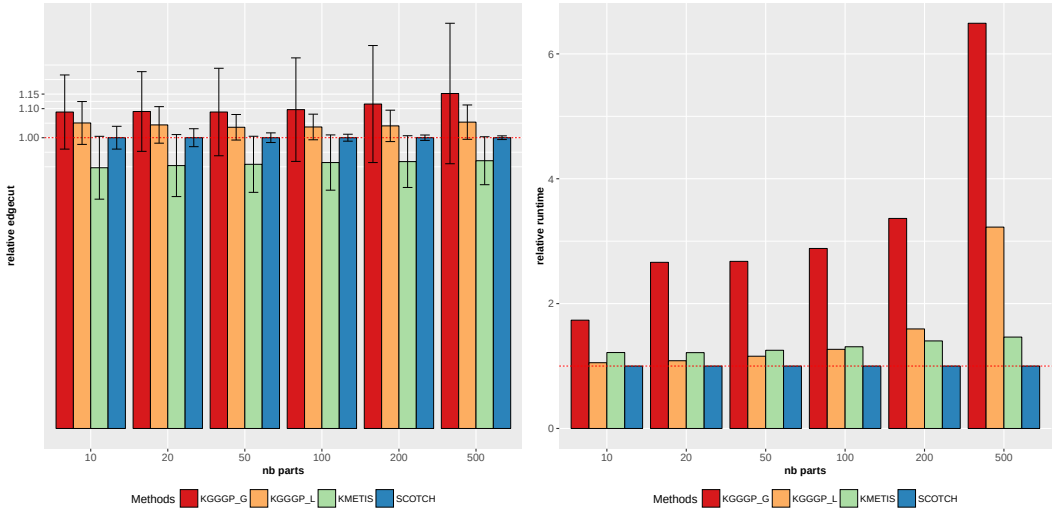
Figure 3.9: KGGGP evaluation on the Walshaw collection relatively to SCOTCH, without multilevel framework. Comparison of the three gain formulas (*classic, diff* and *hybrid*) for a parameter α equal to 1 or 10, and comparison of the *global* and *local* version of KGGGP.

3.5.2 Experiment without Fixed Vertices

In this experiment, we compare both versions of KGGGP (KGGGP_G and KGGGP_L) to SCOTCH and KMETIS on graphs that do not contain any initially fixed vertices. The purpose of this experiment is to evaluate the overall performance of KGGGP compared to two of the best partitioning tools. Clearly we do not expect that KGGGP will outperform SCOTCH or KMETIS for any metric (edgecut or runtime). This is because both KMETIS and SCOTCH use the RB paradigm which is globally recognized as the best approach for the classic graph partitioning. What we aim to find out through this experiment is the relative increase of edgecut obtained by KGGGP algorithms for problems with no fixed vertices.

In Figure 3.10a we present the edgecut results after average calculations over the entire *DIMACS'10* collection, as the number of parts increases. Here, we clearly see that KMETIS provides the best partitioning quality followed by SCOTCH with an average edgecut increase of 9%. This confirms that RB based methods perform better than the greedy ones for the classic graph partitioning problem. Following, the edgecut increase of KGGGP_L is on average 4% compared to SCOTCH and thus 13% compared to KMETIS. Finally, KGGGP_G has an average edgecut increase of 10% compared to SCOTCH and thus 19% compared to KMETIS (the average here is assumed over the different number of parts). Moreover, in Figure 3.10b we present the average execution time of

each method over the entire collection, as the number of parts increases. Here, one may see that Scotch has the best performance followed by kMetis or kgggp_l depending on the number of parts. Additionally, we observe that the execution time of both kgggp_g and kgggp_l become worse as the number of parts increases. This is not surprising if we recall their time complexity compared to RB based algorithms. However, it is important to remark that kgggp_l manages to reduce the execution time compared to kgggp_g and is two times faster than the latter.



(a) Edgecut (b) Runtime

Figure 3.10: Average results over the entire *DIMACS'10* collection for graphs without fixed vertices.

To complete the analysis of this experiment, in Figure 3.11 we present the same results but we depict them separately for each group of graphs. Here we may see that for certain collections of graphs (Matrix, Numerical, dynframe and Walshaw), the edgecut of kgggp_l and kgggp_g is almost the same as that of Scotch. More precisely, if we focus just on the results of kgggp_l, we see that the edgecut for these groups is on average 7% more than that of kMetis and 2% more than that of Scotch. Note that these groups include either graphs that derive from numerical simulations (Matrix, Numerical and dynframe) or graphs that are deliberately chosen to test partitioning methods (Walshaw). In the remainder of this chapter, we will refer to this group of graphs as *group1*. On the other hand, graphs from Clustering or Streets collections appear to be more difficult to partition. Note that these graphs have a particular structure with vertices of highly varying degrees (Clustering) or many vertices that are connected to only one vertex (Streets). This group of graphs is denoted as *group2*. To conclude, we believe that the optimized

version of KGGGP (KGGGP_L) may be a fine partitioning choice even for classic graph partitioning problem, depending on the characterists of the graph in use.

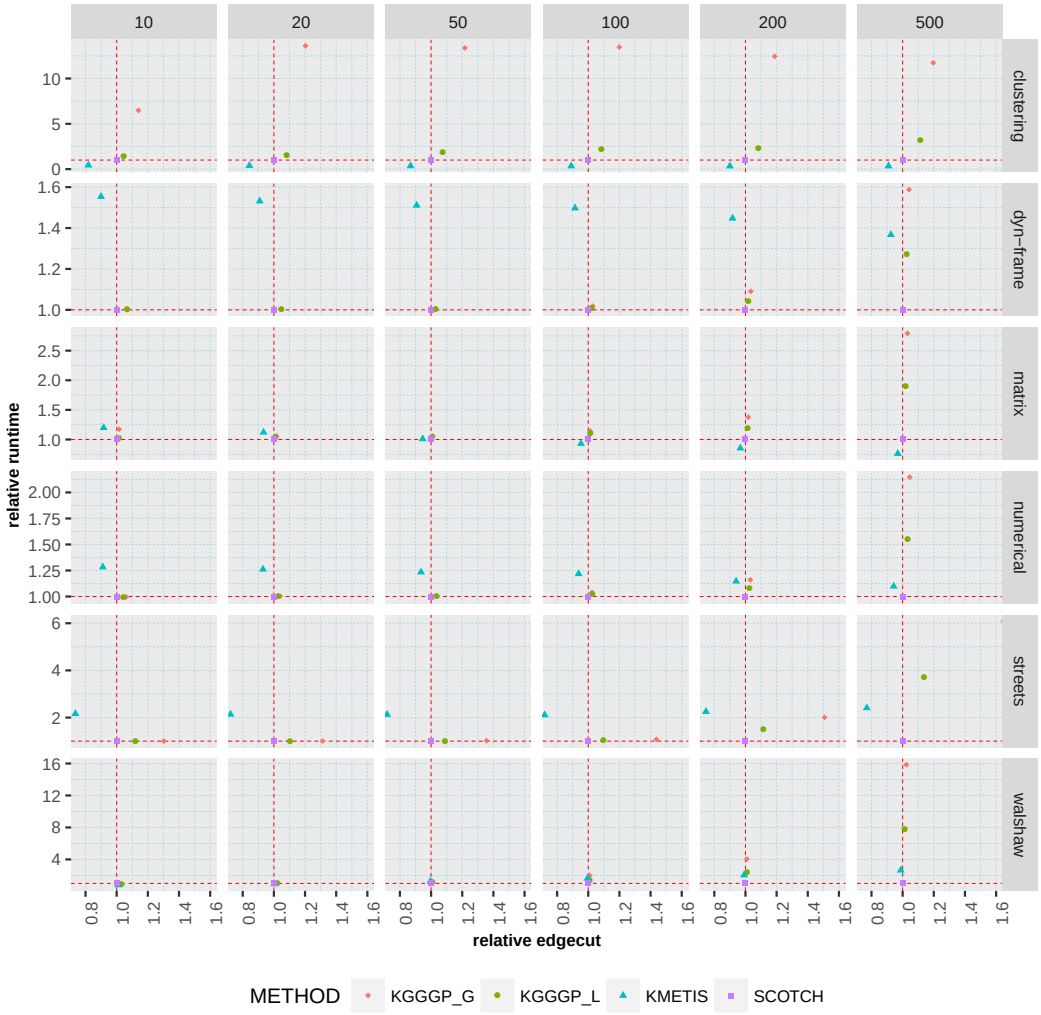


Figure 3.11: Average results on the edgecut quality and time execution for each group in the *DIMACS'10* collection (without fixed vertices).

3.5.3 Experiments with Fixed Vertices

Following, we present results with fixed vertices from two experimental cases, each one representing a different way to distribute the initial fixed vertices to the graph before the partitioning. We believe that the two proposed experiments represent configurations of fixed vertices that may appear in real life

problems in areas such as circuit design or dynamic load balancing. We denote these schemes *bubble* and *repart*.

In this section, we compare graph partitioning methods that support initially fixed vertices such as KGGGP, RBBGM and the RB-based method of SCOTCH. Note that KMETIS does not support fixed vertices so it is not part of the following experiments. Also, based on the previous experiment, we decide to present results only for the KGGGP_L method and not for KGGGP_G. Indeed KGGGP_L has been proven to be faster than KGGGP_G while it produces partitions with similar quality.

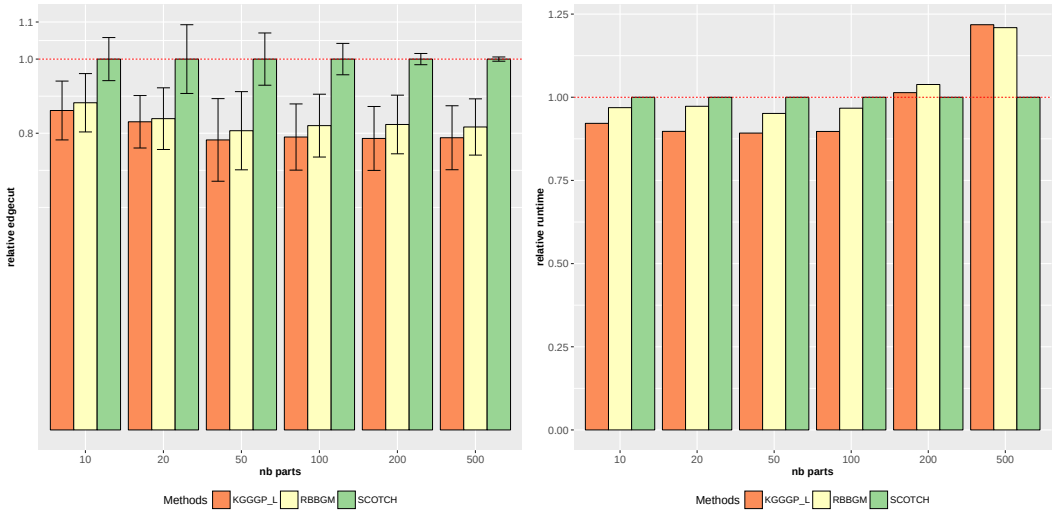
Bubble Scheme

In the *bubble* scheme, we simply compute k initial seeds based on a BFS [25] technique and we use each one as the center of a bubble of fixed vertices. In particular, each bubble is a group of fixed vertices initially assigned to the same part that grows until it reaches a certain percentage of ideal vertex weight. In this scheme, we allow different weights among the bubbles that vary linearly from 5% to 20% of the ideal part size.

In Figure 3.12a, we depict the average edgecut results over the entire *DIMACS'10* collection for all methods involved in this experiment. Here, one may see that both KGGGP_L and RBBGM reduce the edgecut compared to SCOTCH with an average gain of 19% and 16% respectively. Let us now examine the above results in respect to *group1* and *group2* separately. In Figure 3.13a we depict the average edgecut results just over the graphs that belong to *group1*. Here, we see that both KGGGP_L and RBBGM equally minimize the edgecut with an average gain of 19%. Now if we focus on the edgecut results obtained for *group2* in Figure 3.13b, we may see that KGGGP_L provides better results with an average gain of 20% followed by RBBGM with a gain of 10%.

To measure the performance results of this experiment, in 3.12b we present the average execution time of each method over the entire collection. In this figure we see that KGGGP_L and RBBGM perform sightly better than SCOTCH until the number of parts reaches 200, but become worse than SCOTCH when the number of parts is 500. If we focus on the performance results of *group1* and *group2* separately (in Figures 3.13c and 3.13d), we may see that the overhead of KGGGP_L is due to *group1*. More precisely in Figure 3.14, it is clear that the poor performance of KGGGP_L comes mainly from the Walshaw collection where KGGGP_L is 2.5 times worse than SCOTCH when the number of parts is 500.

This overhead is related to the fact the KGGGP_L fails to find a balanced partition in 10% of the total executions for the Walshaw collection. Note also that both SCOTCH and RBBGM have difficulties finding a partitioning solution and often exhibit a large number of failures. In Table 3.3 we demonstrate the percentage of failures for each method when the number of parts is 500. In



(a) Edgecut. (b) Runtime.

Figure 3.12: Average results over the entire *DIMACS'10* collection (*bubble* scheme).

this context, we may say that KGGGP_L is more robust since it fails only for the Walshaw group.

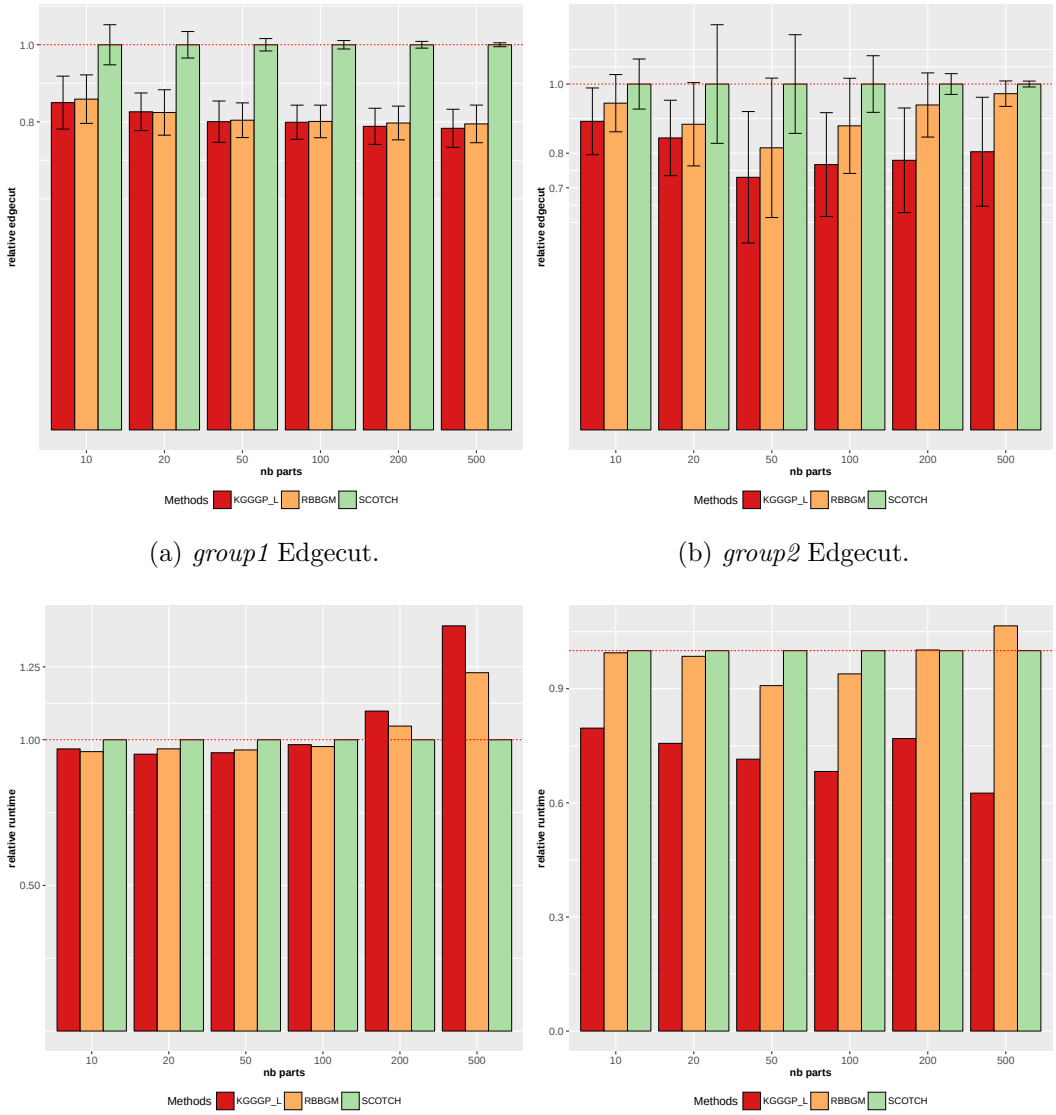
Table 3.3: Failure percentages of each method when the number of parts is 500 (*bubble* experiment).

method	streets	matrix	numerical	clustering	dynframe	walshaw
SCOTCH	88%	2.5%	32%	33%	10%	62%
RBBGM	97%	7.5%	45%	40%	5%	60%
KGGGP_L	0%	0%	0%	0%	0%	10%

Repartitioning Scheme

For the *repart* scheme, we follow the repartitioning method proposed by ZOLTAN in [20] (and described in Section 3.1), where an initial partition is used and its total vertex weight is randomly modified in order to obtain 50% of load imbalance. Based on the above imbalanced partition, we build an enriched graph adding one single fixed vertex per part along with the migration edges that connect it with its respective part. Note that the enriched graphs are larger in size compared to the original ones, thus we test our algorithms on a sub-collection of the graph set in Table 3.4 omitting the large collections dynframe and Streets.

For this experiment, global results on the edgecut and execution time are presented in Figures 3.15a and 3.15b while the same results are also depicted



(a) *group1* Edgecut.

(b) *group2* Edgecut.

(c) *group1* Runtime.

(d) *group2* Runtime.

Figure 3.13: Average results over *group1* and *group2* (*bubble* scheme).

separately for each group of graphs in 3.16. In Figure 3.15a, we see that KGGGP_L obtains minimized edgecut results for the entire collection with an average gain of 11% compared to SCOTCH and 18% compared to RBBGM. More precisely, if we examine the results of each group of graphs separately, we notice that KGGGP performs very well for graphs that come from numerical simulations. For instance, KGGGP_L exhibits an edgecut minimization of up to 30% compared to SCOTCH and up to 36% compared to RBBGM for the Numerical group, as seen in Figure 3.16. Additionally, we observe that RBBGM does not produce partitions of high quality and thus may not be a suitable solu-

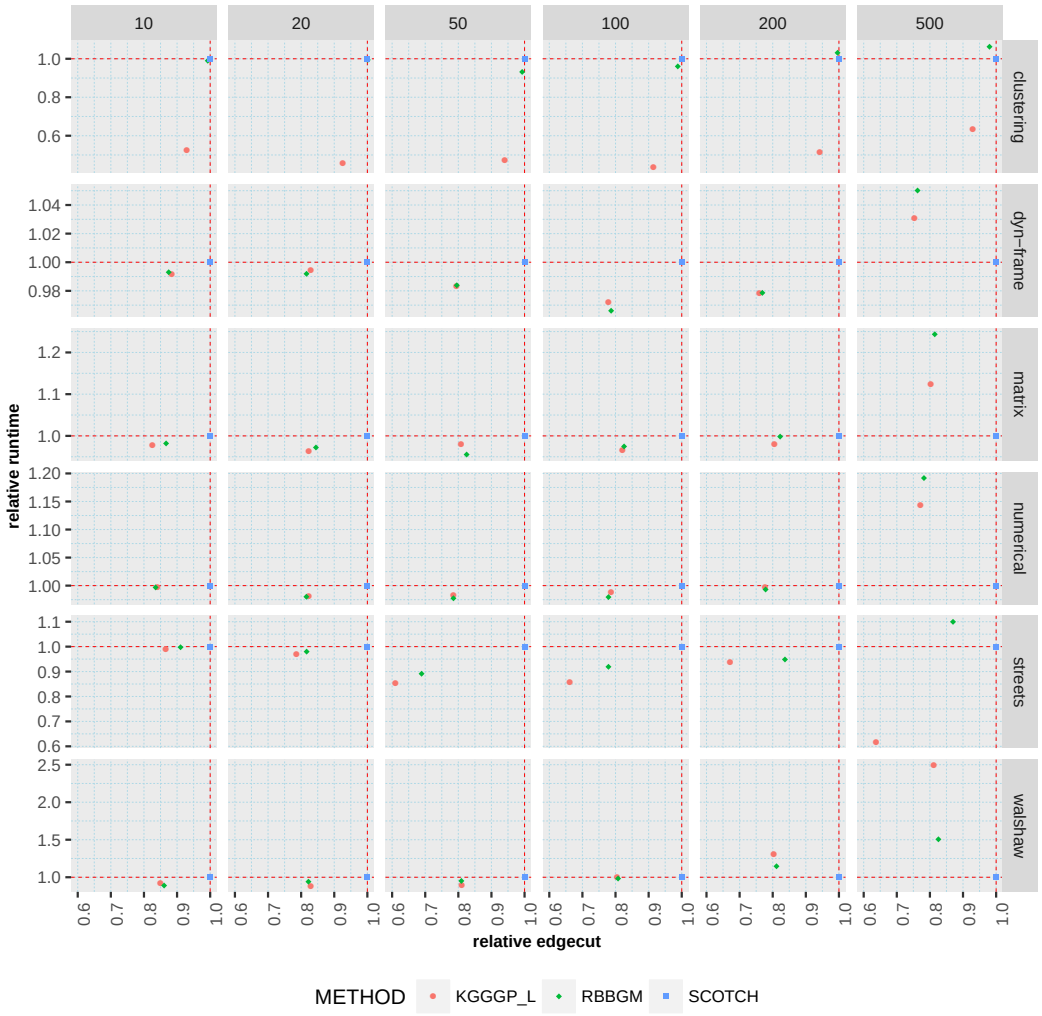
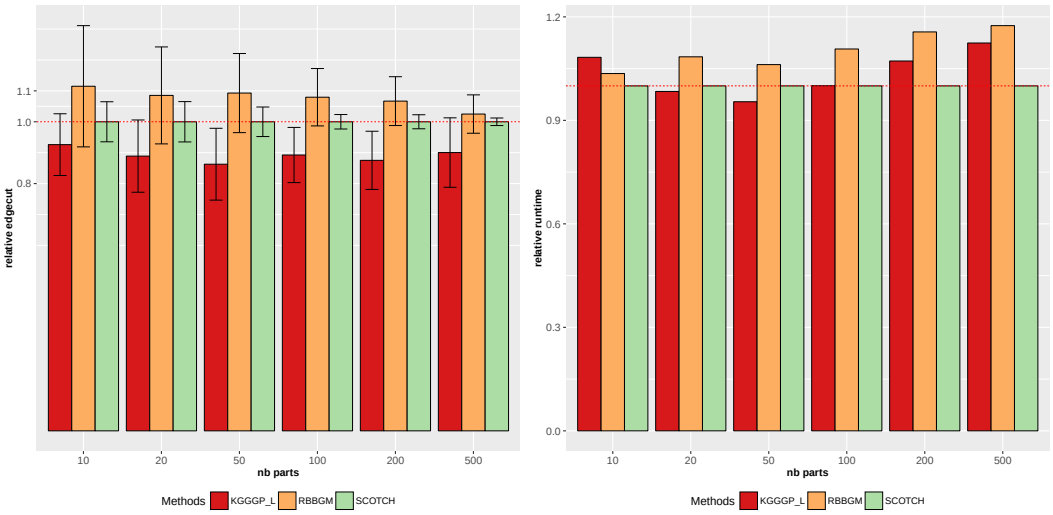


Figure 3.14: Average results on the edgecut quality and time execution for each group in the *DIMACS'10* collection (*bubble* scheme).

tion for the *repart* scheme. A possible explanation is that, in this experiment, fixed vertices are connected through migration edges to all vertices of their respective part. As a result, the maximum-weight graph matching problem that should be solved during this method is more complicated. As a result the additional edgecut of re-introducing fixed vertices to the graph is rather significant.

Finally, regarding the runtime performance of this experiment, in Figure 3.15b one may see that Scotch is often the fastest method along with kggp_l while rbbgm is always the slowest one. Even though kggp is not much slower than Scotch, these results confirm the poor performance

of KGGGP_L as the number of parts increases. However for certain groups of graphs like Numerical and Matrix, we may see in Figure 3.16 that KGGGP_L is the fastest method for all number of parts.



(a) Edgecut. (b) Runtime.

Figure 3.15: Average results over the entire *DIMACS'10* collection (*repart* scheme).

3.5.4 Experiments with Hypergraph Tools

Finally, in order to further evaluate our results, we add a last comparison between the above methods and two well-known hypergraph tools, PaToH and Zoltan. We choose these hypergraph tools because they are quite efficient and they support problems with initially fixed vertices (as seen in Table 3.1). Note that partitioning a graph structure with hypergraph tools may be somewhat unfair. This is due to the fact that hypergraph tools are designed for more complicated structures and an additional overhead is introduced when they partition a graph. However it is still interesting to compare the results especially in terms of edgecut minimization.

To convert the graphs to hypergraphs, we transform each edge of the graph to an hyperedge of size two. In order to conduct the above experiment, we use a small collection, for instance the Walshaw collection. Again, we perform experiments with fixed vertices following the *bubble* and *repart* schemes. For the hypergraph tools we use the *connectivity-1* metric (mentioned in Section 2.2) to measure communications because it can directly be compared to the classic edgecut metric for graphs. Note that in hypergraph partitioning, other metrics may be more important such as the total or maximum communication volume.

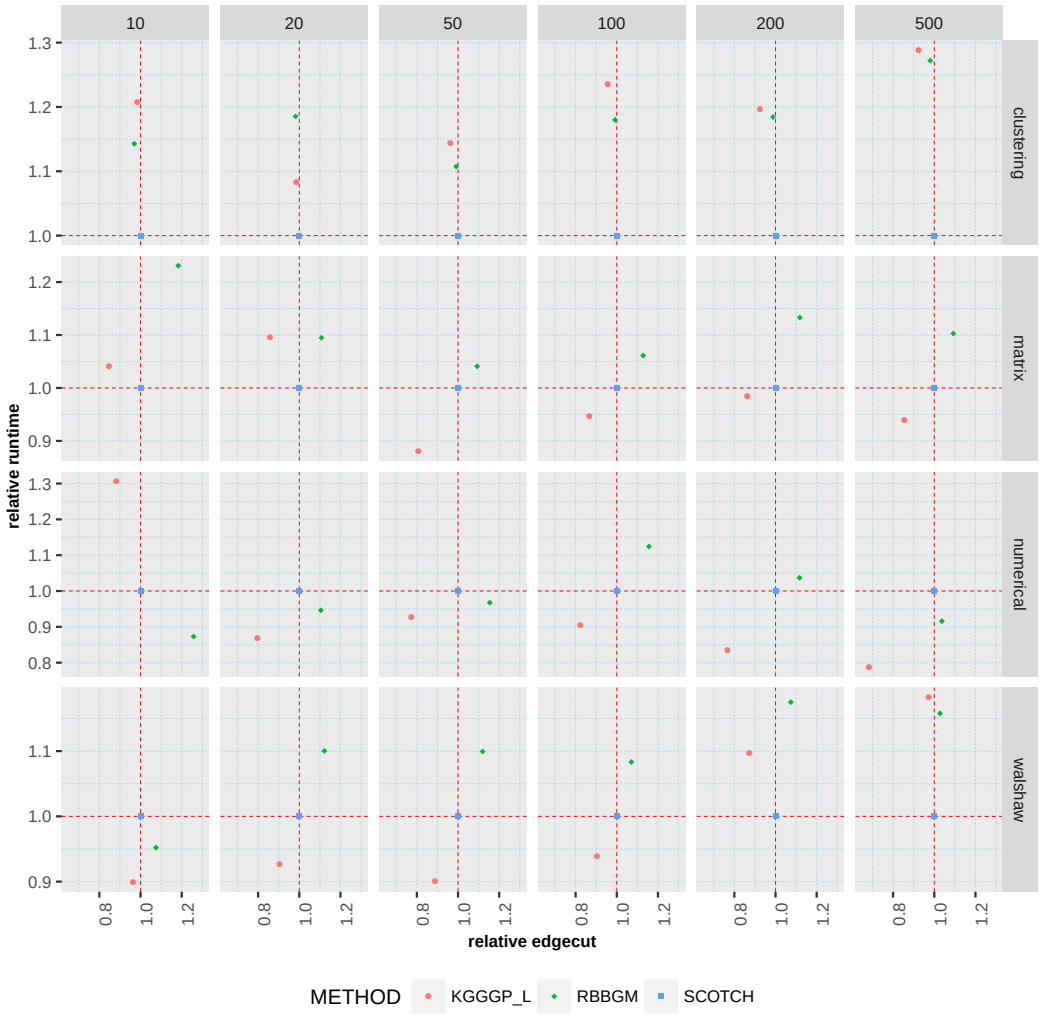


Figure 3.16: Average results on the edgecut quality and time execution for each group in the *DIMACS'10* collection (*repart* scheme).

However since graph partitioning algorithms primarily minimize the edgecut metric, we continue to use it as the main metric for these experiments.

The edgecut results for these experiments are depicted in Figures 3.5.4 and Figure 3.5.4 for the *bubble* and *repart* schemes respectively. For the *bubble* scheme, Zoltan produces partitions of slightly better quality compared to Scotch with an edgecut gain of 2%. For the *repart* scheme the edgecut results of Zoltan are almost the same as that of Scotch. Finally, PaToH does not manage to minimize the edgecut in these experiments.

Here we choose not to show the performance results of the hypergraph tools, since such a comparison would be unfair. During both experiments

(*bubble* and *repart*), PaToH and Zoltan were the slowest tools. This result is expected, since PaToH and Zoltan use complicated structures designed for hypergraphs that are treated by more time consuming partitioning algorithms. Also Zoltan is meant to be a parallel partitioner, thus using it here as a sequential tool introduces a huge overhead. However, we may remark that PaToH scaled better than the other methods as the number of parts increases.

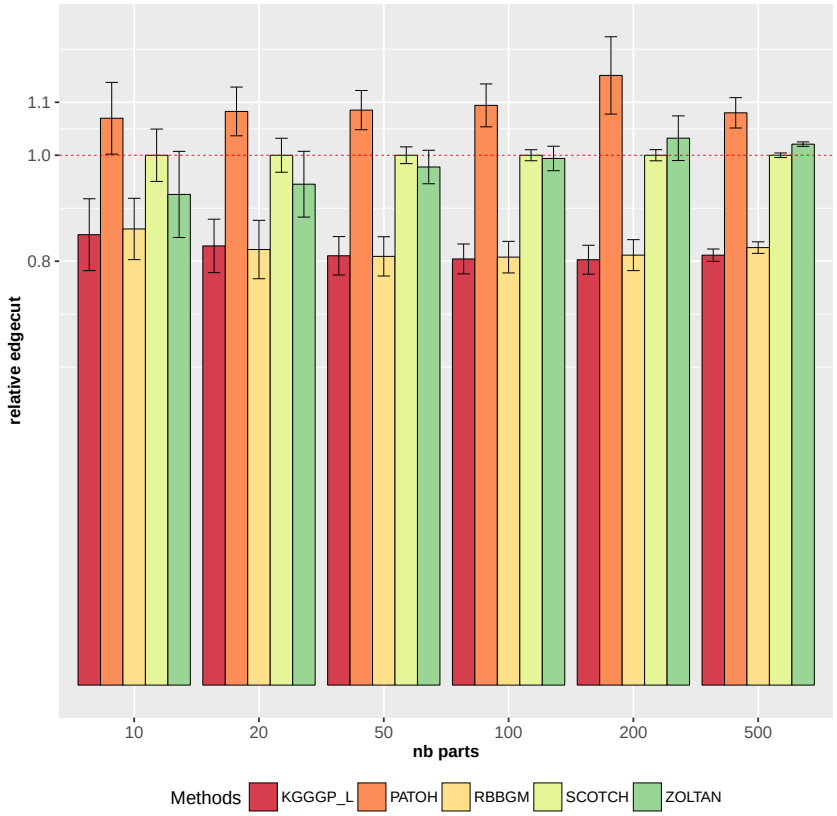


Figure 3.17: Average edgecut results over the Walshaw collection, comparing graph algorithms to hypergraph tools for the *bubble* scheme.

3.6 Conclusion

This work is driven from our interest in graph partitioning with initially fixed vertices that appear in many scientific problems such as the dynamic load balancing of large-scale applications. We strongly believe that the above problem plays an important role in solving the load balancing of coupled simulations. Under this model, additional constraints of the underlying problem are represented in the graph with initially fixed vertices. In this case, we notice that the state-of-the-art algorithm (RB), often does not produce partitions of good

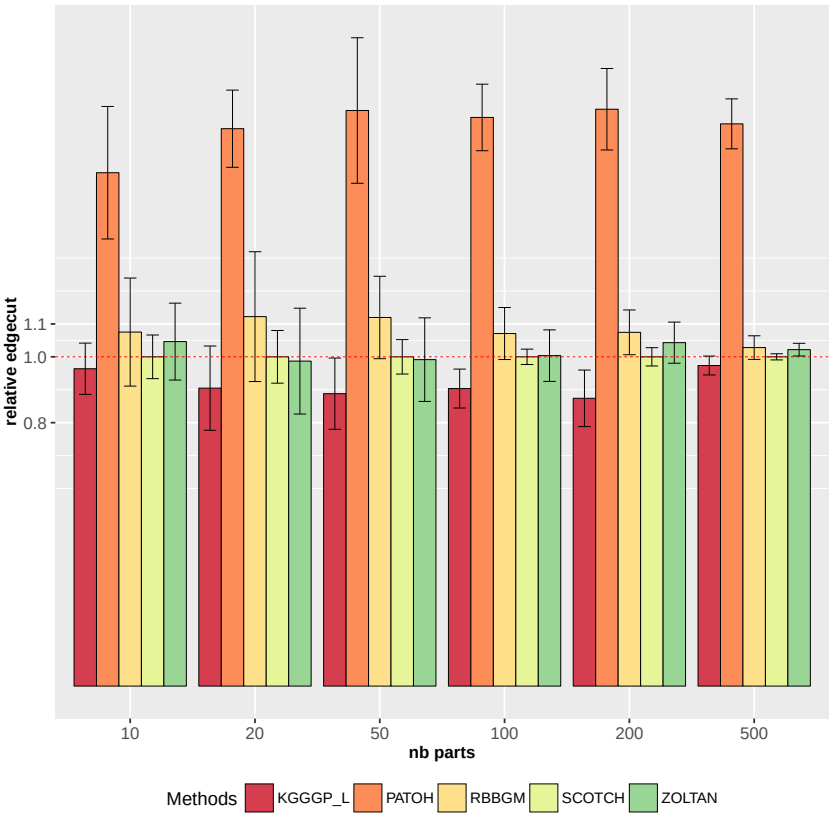


Figure 3.18: Average edgecut results over the Walshaw collection, comparing graph algorithms to hypergraph tools for the *repart* scheme.

quality. Here, we investigate the behavior of RB methods under this constraint and we present a comparison between RB and the two main alternative algorithms for such problems. More precisely, the first alternative is KGGGP, a new greedy graph growing algorithm and the method proposed in KPATOH. In this chapter we compare the optimized version of KGGGP (KGGGP_L) and an implementation of KPATOH for graphs, named RBBGM with an RB-based algorithm (SCOTCH). This implementation was necessary since KPATOH is not publicly available and allows a fair comparison of the algorithms that support problems with fixed vertices.

In this study, KGGGP_L and RBBGM are tested on two different configurations of fixed vertices and their results in terms of edgecut minimization and runtime performance indicate that they handle graphs with fixed vertices better than RB. More precisely, in the first experiment both methods exhibit an up to 25% edgecut minimization, while in the second one KGGGP performs better than RBBGM and SCOTCH, with a maximum gain of 36% and 30% respectively. Note finally that KGGGP remains robust to different graph structures compared to RBBGM and SCOTCH but is amenable to a high number of

parts in terms of runtime performance.

Table 3.4: List of graphs used for experiments from the popular DIMACS'10 collection.

group	collection	graph	# vtx	# edges	avg d°	min d°	max d°
1	walshaw	144	144,649	1,074,393	14.86	4	26
1	walshaw	4elt	15,606	45,878	5.88	3	10
1	walshaw	cs4	22,499	43,858	3.90	2	4
1	walshaw	cti	16,840	48,232	5.73	3	6
1	walshaw	fe_4elt2	11,143	32,818	5.89	3	12
1	walshaw	fe_ocean	143,437	409,593	5.71	1	6
1	walshaw	fe_sphere	16,386	49,152	6.00	4	6
1	walshaw	whitaker3	9,800	28,989	5.92	3	8
1	walshaw	wing	62,032	121,544	3.92	2	4
1	walshaw	auto	448,695	3,314,611	14.77	4	37
1	matrix	audikw1	943,695	38,354,076	81.28	20	344
1	matrix	ecology1	1,000,000	1,998,000	4.00	2	4
1	matrix	thermal2	1,227,087	3,676,134	5.99	2	10
1	matrix	af_shell10	1,508,065	25,582,130	33.93	14	34
1	matrix	G3_circuit	1,585,478	3,037,674	3.83	1	5
1	matrix	nlpkkt120	3,542,400	46,651,696	26.34	4	27
1	matrix	ldoor	952,203	22,785,136	47.86	27	76
1	matrix	cage15	5,154,859	47,022,346	18.24	2	46
1	numerical	NACA0015	1,039,183	3,114,818	5.99	3	10
1	numerical	333SP	3,712,815	11,108,633	5.98	2	28
1	numerical	NLR	4,163,763	12,487,976	6.00	3	20
1	numerical	adaptive	6,815,744	13,624,320	4	2	4
1	numerical	AS365	3,799,275	11,368,076	5.98	2	14
1	numerical	M6	3,501,776	10,501,936	6.00	3	10
1	numerical	channel-b050	4,802,000	42,681,372	17.78	6	18
1	numerical	venturiLevel3	4,026,819	8,054,237	4.00	2	6
1	dynframe	hugebubbles-00000	18,318,143	27,470,081	3.00	2	3
1	dynframe	hugebubbles-00010	19,458,087	29,179,764	3.00	2	3
1	dynframe	hugebubbles-00020	21,198,119	31,790,179	3.00	2	3
1	dymframe	hugetrace-00000	4,588,484	6,879,133	3.00	2	3
1	dymframe	hugetrace-00010	12,057,441	18,082,179	3.00	2	3
1	dynframe	hugetrace-00020	16,002,413	23,998,813	3.00	2	3
1	dynframe	hugetric-00000	5,824,554	8,733,523	3.00	2	3
1	dynframe	hugetric-00010	6,592,765	9,885,854	3.00	2	3
2	clustering	citationCiteseer	268,495	1,156,647	8.62	1	1318
2	clustering	coAuthorsCiteseer	227,320	814,134	7.16	1	1372
2	clustering	coAuthorsDBLP	299,067	977,676	6.54	1	336
2	clustering	coPapersCiteseer	434,102	16,036,720	73.88	1	1188
2	clustering	coPapersDBLP	540,486	15,245,729	56.41	1	3299
2	clustering	as-22july06	22,963	48,436	4.22	1	2390
2	streets	asia	11,950,757	12,711,603	2.13	1	9
2	streets	belgium	1,441,295	1,549,970	2.15	1	10
2	streets	germany	11,548,845	12,369,181	2.14	1	13
2	streets	great-britain	7733822	8156517	2.11	1	8
2	streets	italy	6,686,493	7,013,978	2.10	1	9
2	streets	netherlands	2,216,688	2,441,238	2.20	1	7
2	streets	luxembourg	114,599	119,666	2.09	1	6

Chapter 4

Partitioning for Coupled Simulations

In this chapter, we focus on the problem of load balancing for coupled simulations, as presented earlier in this work (Chapter 2). Remember that when coupling operations are executed in parallel (following the parallel CCS or the DCS), the computational load during couplings is unevenly distributed among processors and the inter-component communications are not minimized. Under this context, the currently used method that includes classic graph partitioning techniques that balance each component separately, is not sufficient.

In this chapter, we propose new graph partitioning algorithms that address this problem. The idea here is to efficiently partition the data of the coupled simulation as part of an interactive system and not just as part of independent components. In this way, the load is balanced both during internal computations of each component and during the coupling. Additionally, with a partitioning method that is aware of the coupling process, the communication costs between processors of different components may be reduced in terms of the number of exchanged messages. We call the procedure of finding such data distributions the *co-partitioning*. Finally, note that solving the problem of graph partitioning with initially fixed vertices (as studied in Chapter 3) is a crucial step in designing efficient co-partitioning algorithms.

In the remainder of this chapter, we first describe the general model of coupled simulations and then, we formally introduce the co-partitioning problem, extending the classic definition. Furthermore, we propose new co-partitioning techniques that explicitly take into consideration the coupling process and perform a coupling-aware partitioning of the entire simulation. Finally, we validate the proposed methods with a series of experiments on synthetically generated and real-life data.

4.1 Model of Coupled Simulations

In this section, we introduce a new model that describes the general execution of coupled simulations following the organizing principles for coupling in multi-physics and multi-scale simulations, in [47].

A coupled model consists of a number of component models (or simply components), namely $C_i, i = 1, \ldots, N$, that collectively represent a complex system through their evolution and mutual interactions, under a coupling environment. A component C_i represents a model that solves an individual system defined in a computational domain Γ_i. Two components C_i, C_j interact, and thus are coupled, when parts of their entire computational domains overlap, imposing data dependencies on their models. The overlapping creates a common *coupling interface* $\Omega_{ij} = \Gamma_i \cap \Gamma_j \neq \emptyset$. In addition, solutions on the domain of C_i may serve (directly or after computations) as input data for C_j and/or vice versa. Note that the type of domain overlapping may range from the simplest case of a lower dimension surface coupling (in Figure 4.1) to partially volumetric coupling (in Figure 4.1) or to full volume coupling, where $\Gamma_i = \Gamma_j$. In principle, three or more domains can intersect forming high order coupling interfaces, but in this work we assume that each coupling involves exactly two components.

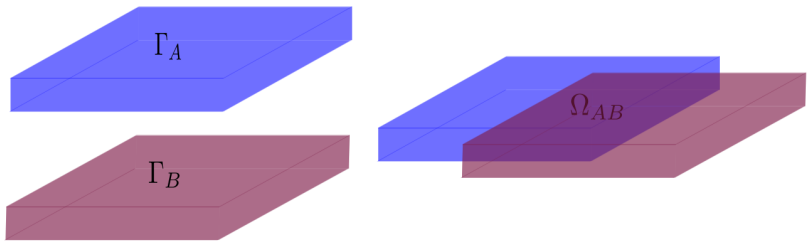


Figure 4.1: Example of coupling interface for two rectangular computational domains.

In some simulations, coupling between components may occur regularly in time with a certain coupling period related to the physical timestep of the coupled simulation. Therefore, depending on the frequency of the coupling, the potential improvement introduced by a co-partitioning algorithm may result in a significant minimization of the total elapsed time. On the other hand, when the coupling is sporadic or highly unpredictable the performance gain may be insignificant or even negative due to the overhead of the co-partitioning.

In Figure 4.2, we illustrate the model of a coupled simulation taken on one iteration with two components running concurrently. This model may be extended to N components, but for ease of presentation we depict it here just for two components, C_1 and C_2. As one can see, a coupled model evolves iteratively in time, entering a sequence of *regular* and *coupling* phases. Note

that, we assume the presence of synchronization in our model, that occur before and after entering a coupling phase.

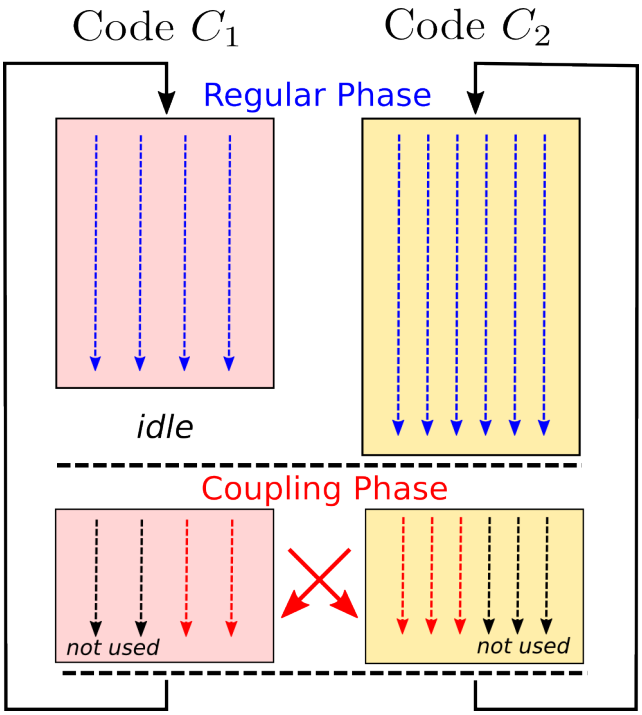


Figure 4.2: General model of a complex simulation with two coupled components C_1 and C_2.

A regular phase occurs when components compute solutions independently, on their own computational domain. In this case, all data associated with each component participate in the computations. The number of processors assigned to each component C_i in the regular phase is k_i. During a regular phase, each component uses a different time and space discretization, so the number of processors required by each solver may vary, and therefore $k_i \neq k_j$ for $i \neq j$. That usually leads to differences in execution time between components at the end of the regular phase, which results in an idle time T_{idle} for each coupling $(C_i \wedge C_j)$. As explained in Chapter 2, T_{idle} denotes the time the fastest component waits in the synchronization step until the slower component finishes calculations on its computational domain and enters a coupling phase.

Besides, a coupling phase occurs when components interact with each other, exchanging data on the coupling interface Ω_{ij} for $i \neq j$. Note that we refer to this type of communication as *inter-component* communication since data are exchanged between processors that belong to different components. Additionally, during this phase the coupling framework drives the overall application, ensuring synchronization among the components, locating data through distributed geometries and performing the interpolation step between different

meshes. Note that the number of active processors for each component during the coupling phase ($k_i^{'}$) is often lower, namely $k_i^{'} \leq k_i$ and depends on the size of coupling interface. Observe that these are the processors of each component that own data on Ω_{ij}.

Algorithm 2 Algorithmic description of a coupled simulation.

Input: C, k, N, t
Input/Output: s
1: **for each** iteration t **do**
2: | **for** component $C_i \in [1, \ldots, N]$ **do**
3: | | **for all** processor k_i **do in parallel**
4: | | | compute($s_i(t)$)
5: | | | internal-exchange($s_i(t)$)
6: | | **end for**
7: | **end for**
8: | wait()
9: | **for each** coupling $(C_i \wedge C_j)$ **do**
10: | | **for all** processor $k_i^{'} \cup k_j^{'}$ **do in parallel**
11: | | | compute($s_i(t)$)
12: | | | inter-component-exchange($s_i(t)$)
13: | | **end for**
14: | **end for**
15: **end for**

In Algorithm 2, the algorithmic description of a coupled simulation with N components $(C_i, i = 1, \ldots, N)$ is illustrated. We assume that for each component an independent data distribution of the computational domain has been computed, as part of a preprocessing step. The data are then distributed in the proper number of processors, that is k_i for $i = 1, \ldots, N$. The processors that are active during the regular phase of C_i are denoted as k_i while its processors that are active during the coupling phase $(C_i \wedge C_j)$ as $k_i^{'}$.

Let us now analyze the total execution time of a coupled simulation with two components under the following expression. Again, this expression could be extended to include N components but for simplicity we focus here on just two:

$$T_{total} = max(\alpha_1 * T(C_1), \alpha_2 * T(C_2)) + T^{'}(C_1 \wedge C_2) \quad (4.1)$$

$$= min(\alpha_1 * T(C_1), \alpha_2 * T(C_2)) + T_{idle} + T^{'}(C_1 \wedge C_2) \quad (4.2)$$

where:

- α_i is the number of iterations performed during the regular phase for C_i, where $i = 1, 2$;
- $T(C_i)$ is the time spent by C_i during the regular phase;
- T_{idle} is the time the fast component has to wait before entering a coupling phase;

- $T^{'}(C_i \wedge C_j)$ is the time spent during the coupling phase between C_i and C_j.

More precisely, since there is an independent data distribution for each component that assigns the computations of each domain to a number of processors, $T(C_i)$ may be written as following:

$$T(C_i) = (T_{comp}(k_i) + T_{comm}(k_i)) \tag{4.3}$$

where:

- $T_{comp}(k_i)$ is the time required for C_i to execute one iteration of computations during the regular phase, taken on the slowest processor;

- $T_{comm}(k_i)$ is the communication time of exchanging data among the processors of C_i during the regular phase.

Therefore, in order to minimize the total execution time of a coupled simulation under a parallel environment, one should minimize each factor of the equation 4.2. Now if we consider that the load balancing of C_1 and C_2 is solved independently with a graph partitioning for each component, the factors $T(C_1)$ and $T(C_2)$ are minimized. In other words, the computations during the regular phase are equally distributed among the available processors for each component and the internal communication costs are minimized based on the edgecut criterion. However, using such a load balancing approach (naive co-partitioning), the T_{idle} is not minimized and neither is the time spent during the coupled phase ($T^{'}(C_1 \wedge C_2)$).

Therefore, in order to minimize the total execution time of a coupled application, we identify two important subproblems:

i. *The resource distribution problem:* Here, the problem is to find a good distribution of the total number of available processors among the components, in order to minimize T_{idle}. The main idea is to assign less processor resources to the fastest component following an empiric approach based on performance studies, as they do for instance in [36]. That way, T_{idle} is minimized and the load balancing of the regular phase for the whole coupled simulation is optimized.

ii. *The data distribution problem:* Assuming a given number of processors for each component (by solving the previous problem), the goal here is to find a good data distribution that minimizes the execution time during the coupling phase. Employing a classic load balancing algorithm for each component separately does not minimize the total execution time of a coupled simulation 4.2, but only the time of each component spent during the regular phase ($T(C_i)$).

More precisely, the time spent during the coupling phase follows the expression below:

$$T^{'}(C_1 \wedge C_2) = max(T^{'}(C_1), T^{'}(C_2)) + T^{'}_{inter-comm}(k^{'}_1 \wedge k^{'}_2) \quad (4.4)$$

where

$$T^{'}(C_i) = T^{'}_{comp}(k^{'}_i) + T^{'}_{comm}(k^{'}_i)$$

and $T^{'}_{inter-comm}(k^{'}_1 \wedge k^{'}_2)$ represents the inter-component communications. Note that $T^{'}_{comp}(k^{'}_i)$, $^{'}T_{comm}(k^{'}_i)$ and $T^{'}_{inter-comm}(k^{'}_1 \wedge k^{'}_2)$ are not minimized by a classic load balancing algorithm since neither the computations nor the communications required during the coupled phase have been explicitly considered during the load balancing. Remember that $k^{'}_i$ is the number of processors that own data belonging to C_i and are active during the coupling phase. Note that under a naive co-partitioning, there is no control on this number since a classic partitioning algorithm ignores the additional requirements (in terms of computations or communications) of each component during the coupled phase.

In this work, we assume that the resource distribution problem is already addressed, and thus T_{idle} is minimized. As a result, we focus on the data distribution problem and we propose algorithms that aim at minimizing the time spent during the coupling phase. Note that there is a trade-off between the minimization of the coupling phase and the minimization of the regular phase which is imposed by the additional requirements of the co-partitioning problem.

4.2 Related Work

In this section, we discuss existing work that is related to the load balancing problem for coupled simulations. Remember that in Chapter 2, we reviewed existing load balancing techniques regarding the resource distribution problem. However note, that as far as we know, partitioning solutions that address the data distribution problem have not been yet proposed in the literature. Nevertheless, in this section, we briefly review studies that address a similar problem, that is, the graph partitioning for *multi-phase* computations.

A multi-phase simulation is a single-constraint code with N distinct computational phases, each separated by an explicit synchronization step. In general, the amount of computations performed for each element of the mesh is different for different phases. In order to effectively solve such multi-phase computations in parallel, one must partition the mesh such that the computation in each phase is balanced, and the amount of interactions among the different processors in each phase is minimized. Note that the traditional graph partitioning model is not effective in such computations. For example, if we assign

to each vertex a weight that corresponds to the total amount of computations performed by all phases, we will get a partitioning that is not necessarily balanced during each computational phase (due to the explicit synchronization steps).

Such an example is the scientific simulations in which the mesh elements come in contact with each other and are routinely performed in the context of simulations that study vehicle crashes, material deformations, and projectile-target penetration. Typically, the simulation involves localized stress-strain finite element calculations over the entire mesh together with a much more complex contact detection phase over the restricted areas of possible penetration. Most of the existing partitioning algorithms cannot effectively decompose these types of simulations as they ignore the underlying geometry and produce subdomains that result in high-communication overheads during the contact-search phase of the computation. In Figure 4.3 one may see various stages of a compact-impact simulation.

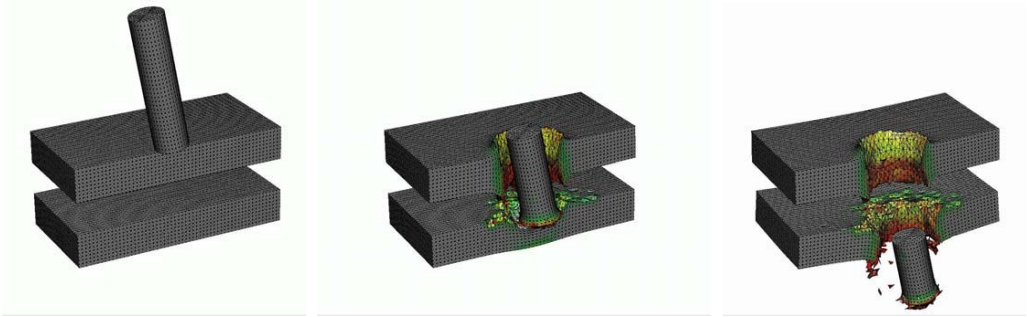


Figure 4.3: Example of contact-impact simulation (extracted from [41]).

A first approach to balance contact-impact simulations is proposed in [55] and uses two different partitions of the mesh, one for the contact phase and another one for the finite element calculation. In the first partitioning, a traditional multilevel graph partitioning algorithm is used to evenly distribute the entire mesh, whereas in the second partitioning, a recursive coordinate bisection (RCB) algorithm is used to evenly distribute only the surface elements. As a result, these two different decompositions ensure that the overall computation is balanced throughout the entire simulation. The main disadvantage of this approach is that data must be redistributed between the two partitions at every time-step and that some memory is duplicated. However, the authors report that the advantage of achieving load-balance in both phases greatly outweighs the cost of maintaining two partitions.

Another approach to solve such problems is based on the multi-constraints partitioning algorithm, initially proposed in [38]. The multi-constraint partitioning method extends the classic graph partitioning problem formulation, by assigning a vector of weights to each vertex, that represents the different

computational load of the vertex in every phase. The problem now becomes that of finding a partitioning that minimizes the communication costs and is balanced for each weight constraint.

However, even though this multi-constraint based approach achieves the desired load balance, care must be taken to ensure that the overall approach does not lead to excessive communication overheads during the contact-impact phase for contact detection. To overcome the above problem, in [41] the authors develop multi-constraint graph partitioning algorithms that take into account the underlying geometry. Additionally, they propose better parallel search algorithms specifically designed for locating detections, that reduce the number of excessive communication. Finally, a last approach that employs the model of the multi-constraint method is presented in [72] and aims to find a partition for the compact-impact simulations that treats each phase separately, using results obtained from previous phases.

At first glance, one could consider the co-partitioning problem as several multi-phase problems, each one representing a different component. However this representation is not adequate, since it does not include the coupling communications between components which indicate external dependencies between different multi-phase simulations. Moreover, since this approach has no control on the number of processors in each phase, it would impose the use of all available processors in the coupling phase, that may degrade the global edgecut. This is especially true if the coupling interface is small compared to the whole computation domain.

4.3 The Co-Partitioning Problem

In this section, we first give a formal definition of the co-partitioning problem, equivalent to the classic partitioning one. Then we introduce some graph operators based upon which we describe our co-partitioning algorithms, called Aware, ProjRepart and ProjSubpart. For reasons of consistency, we also describe the state-of-the-art method called Naive as a sequence of graph operators.

4.3.1 Definition of Co-Partitioning

Here, we explain how we propose to enrich the classic graph model to take into account the coupling phase of a coupled simulation explicitly and we formally define the co-partitioning problem. In the remainder of this chapter, we assume coupled simulations with two components represented by subscript letters A, B while the exponent notation $'$ denotes the coupling phase.

Let us consider that each individual component involved in the simulation, is represented by a graph, that is $G_A = (V_A, E_A)$ for model A and $G_B =$

(V_B, E_B) for B. As shown in Figure 4.4, we represent the coupled model with the *global graph*, $G_{AB} = (V_A \cup V_B, E_A \cup E_B \cup I_{AB})$, where I_{AB} is the set of weighted edges that interconnect some vertices of V_A and V_B. We call those edges *interedges*[1] (as they connect vertices from different graphs), and they represent the communication costs of exchanging data that belong to different components during the coupling phase (inter-component communications). In practice, these interedges are located thanks to geometric intersection between mesh cells of A and B.

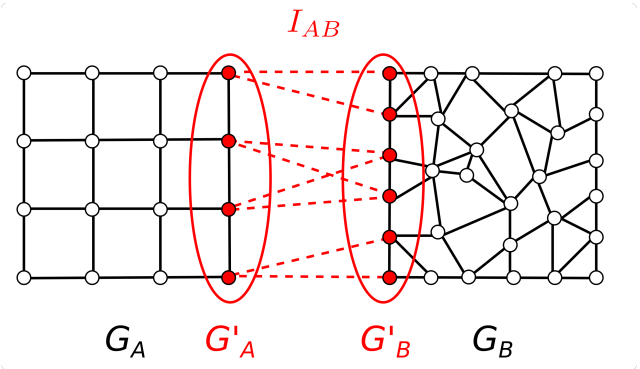


Figure 4.4: Example of a global graph G_{AB} based on two graphs G_A and G_B and interedges I_{AB} (in dashed red), showing the coupled subgraphs G'_A and G'_B (circled in blue).

Moreover, we may define the *coupled graph* G'_{AB} as the subgraph of G_{AB} whose vertex set includes only vertices that are adjacent to interedges. Likewise, we can obtain the *coupled subgraphs* $G'_A = (V'_A, E'_A)$ and $G'_B = (V'_B, E'_B)$ from G_A and G_B respectively.

In this context, we aim to find a partitioning of the global graph that is aware of the coupled subgraph. We call this problem *the co-partitioning problem*. More precisely, a K-way partition P_{AB} of the global graph G_{AB} is said to be a (k_A, k_B, k'_A, k'_B)-way *co-partition*, if the following conditions hold:

- P_A is a k_A-way balanced partition of G_A,
- P_B is a k_B-way balanced partition of G_B,
- P'_A is a k'_A-way balanced partition of G'_A,
- P'_B is a k'_B-way balanced partition of G'_B,
- $P'_A = P_A \setminus V'_A$,
- $P'_B = P_B \setminus V'_B$,
- $P_{AB} = P_A \cup P_B$,

[1] In other words, the interedges I_{AB} just represent a binary relation from V_A to V_B.

- $K = k_A + k_B$,
- $k_A^{'} \leq k_A$ and $k_B^{'} \leq k_B$.

Following the above definition, we expect that such a co-partition will provide a good load balancing for both the regular phase and the coupling phase, and for both components A and B, since it explicitly finds a partition for the graphs that participate in the regular phase (G_A and G_B) and their subgraphs involved in the coupling phase ($G_A^{'}$ and $G_B^{'}$). As an additional criterion, we aim to minimize the inter-component communication cost (represented by the weighted interedges), by minimizing the total number of messages ($totZ$) and the maximum volume of communication per processor that participate in the coupling phase ($maxV$). Note that the total volume of communication ($totV$) remains rather constant in the coupling phase.

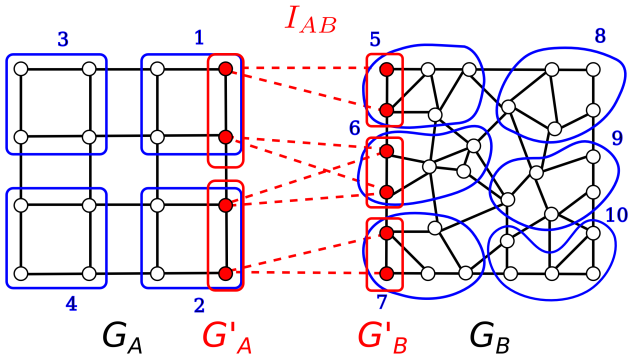


Figure 4.5: Example of $(4, 6, 2, 3)$-way co-partition of G_{AB}.

Figure 4.5 depicts a $(4, 6, 2, 3)$-way co-partition. Assuming all vertex/edge weights are 1, we see that the balance criterion is perfectly respected in the regular phase for both P_A and P_B ($k_A = 4$ and $k_B = 6$), with an edgecut respectively equal to 8 and 12. As for the coupling phase (red vertices), $P_A^{'}$ and $P_B^{'}$ are perfectly balanced as well ($k_A^{'} = 2$ and $k_B^{'} = 3$), with an edgecut equal to 1 and 2 respectively. Considering the coupling communication, $totV = 6$, $maxV = 2$ and $totZ = 4$, corresponding to the following processor pairs assigned to different components: $(1, 5), (1, 6), (2, 6), (2, 7)$. The following communication matrix $C = (C_{i,j})_{k_A \times k_B}$ represents the coupling exchange from A to B:

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Based on C, it is easy to define the coupling metrics:

$$
\begin{aligned}
totV &= \sum_{1\leq i\leq k_A, 1\leq j\leq k_B} C_{i,j} \\
maxV &= \max\left(\max_{1\leq i\leq k_A}\sum_{1\leq j\leq k_B} C_{i,j}\,,\ \max_{1\leq j\leq k_B}\sum_{1\leq i\leq k_A} C_{i,j}\right) \\
totZ &= |\{C_{i,j} > 0\}|
\end{aligned}
$$

4.3.2 Graph Operators

Subsequently, we describe in detail the basic steps of the proposed algorithms. To facilitate this description, we introduce some graph operators as building blocks that produce new graph structures or partitions. Namely, we use the following operators: *partition, repartition, subpartition, restriction, extension* and *projection*.

Partition. First, we define the partition operator as $Part(G, k) \rightarrow P$ which simply returns a k-way balanced partition of the graph G with respect to an imbalance tolerance ϵ (see Chapter 2 for the definition).

Repartition. Then, the repartition operator is defined as $Repart(G, P, k, l) \rightarrow P'$. This operator computes a new l-way balanced partition of a graph G using a former (possibly unbalanced) k-way partition of the same graph, such that the migration volume is minimized as an additional criterion. This repartition operator uses the $k \times l$ repartitioning algorithm presented in [70], that contrarily to classic repartitioning algorithms (scratch-remap [52], diffusion [60, 44] or biased partitioning [5, 20]) enables to change the target number of parts (i.e., $l \neq k$). The $k \times l$ repartitioning algorithm constructs, with a greedy strategy, a good migration matrix[2] C, that minimizes both the total migration volume ($totV$) and the total number of messages exchanged during migration ($totZ$). Then it performs a biased partitioning of the graph G, enriched with l additional fixed vertices connected to the k former parts based on the migration matrix C (Fig. 4.6). More details on this operator and its implementation can be found in [69, 70].

Subpartition. The subpartition operator is defined as $Subpart(G, P, k, l) \rightarrow P'$. Assuming that P is well-balanced, this operator is a simple alternative to the repartition operator, in the particular case where k and l are multiples with $k \leq l$. This operator computes a l-way balanced partition of a graph G using

[2] A migration matrix $C = (C_{i,j})$ of dimension $k \times l$ represents the amount of data that migrates from a *former* part i to a *newer* part j (if $i \neq j$) or the amount of data that remains in place (if $i = j$).

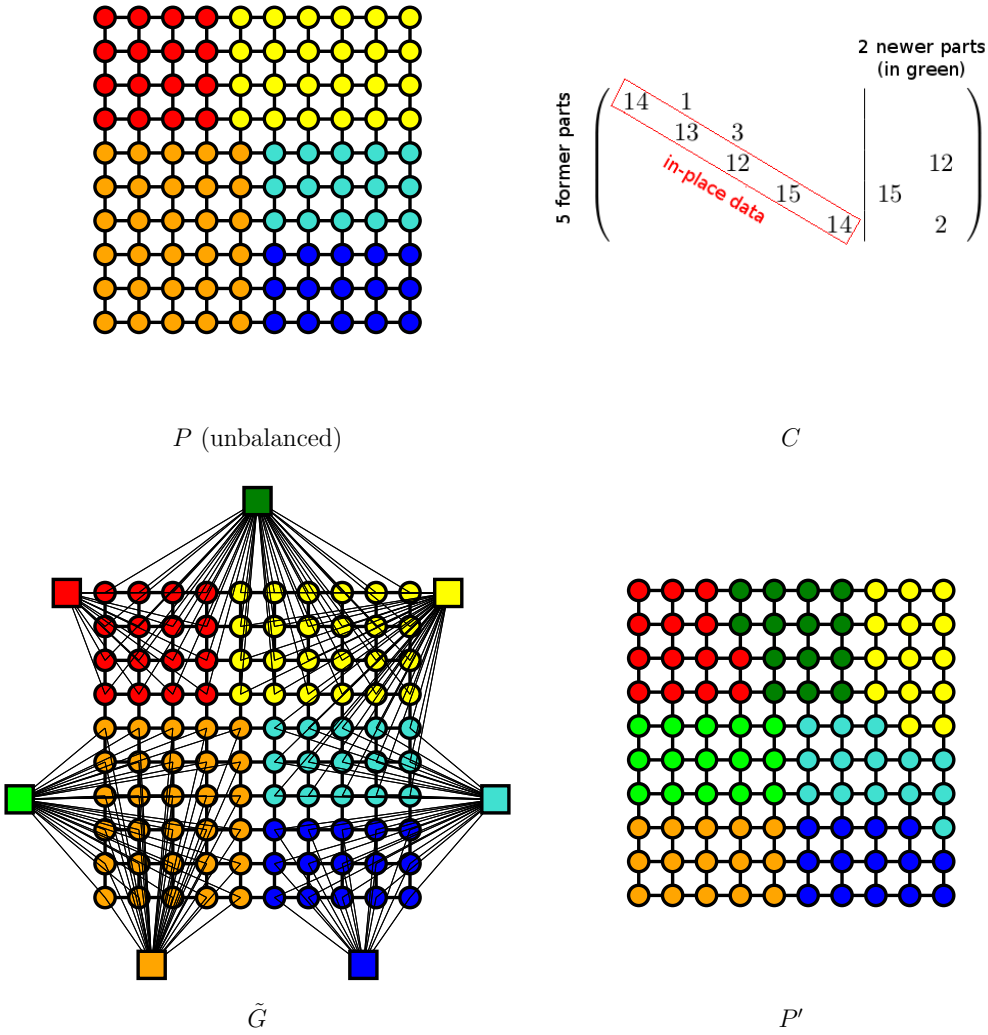
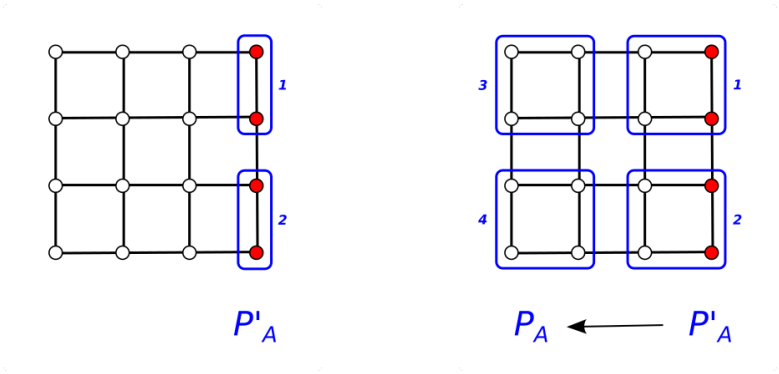


Figure 4.6: Sample of a 5×7 repartitioning of a 2D grid of dimensions 10×10. The migration matrix C explains how vertices will migrate from the 5 former parts to the 7 newer. It is chosen to minimize both $totZ$ and $totV$. On the enriched graph $\tilde{G}$, the square vertices represents the l fixed vertices, connected to former parts according to C. The final partition obtained is well balanced and respects the communication scheme imposed by C.

an initial balanced k-way partition of the same graph (under an imbalance factor ϵ). It just subdivides each part of P into l/k subparts to obtain a final partition P' into l parts. This method is straitghforward, as it just requires to apply the partition operator k times using an imbalance tolerance of $\epsilon/2$.

Restriction. The restriction operator can be expressed as $Rest(G, V') \rightarrow G'$, and returns a subgraph G', that is the restriction of $G = (V, E)$ to vertex set $V' \subset V$. Given the interedges I_{AB} and the graph G_A, we will use this operator to compute the coupled subgraph G'_A as $Rest(G_A, Dom(I_{AB}))$ where $Dom(I_{AB}) = \{u_a \in V_A, (u_a, u_b) \in I_{AB}\}$ is the departure domain of I_{AB}. The Figure 4.14 illustrates this operator on two cubic meshes with a surface coupling.

Extension. Next, we define the extension operator $Ext(G, G', P', k, l) \rightarrow P$, which returns a l-way partition of a graph G using a given k-way partition P' of a subgraph $G' \subset G$, where $l \geq k$. The idea behind this operator is to extend a given partition P'_A of the coupled subgraph G'_A to the graph G_A, such that vertices already assigned to a part in P'_A remain fixed in the new partition P_A, as shown in Figure 4.7.



(a) initial partition (b) extended partition

Figure 4.7: Given the k'_A-way partition P'_A of G'_A in Figure 4.7a, the extension operator computes a k_A-way partition P_A of G_A in Figure 4.7b, with $k'_A = 2$ and $k_A = 4$.

Projection. Finally, we define the projection operator, $Proj(G, G', I, P, k) \rightarrow P'$, that finds a k-way partition of a graph G', using a given k-way partition of a graph G and the interedges $I \subset V \times V'$, with $G = (V, E)$ and $G' = (V', E')$. The key idea of the projection operator is to compute a *similar* partition of G on G' using the relation provided by interedges I, that map vertices from V to V'. More precisely, lets consider a vertex $u' \in V'$ and the vertex set $S(u') = \{u \in V, (u, u') \in I\}$. In the case where all the vertices of $S(u')$ are mapped to the same part p in P, then u' is trivially assigned to this part p in P'. In the case where the vertices of $S(u')$ are mapped to different parts, the situation is ambiguous, and one must select a part for u' according to a second criterion, like the edgecut optimization. In practice, this operator is used to

compute a similar partition between the two coupled subgraphs G'_A and G'_B, connected through interedges I_{AB}, as shown in Figure 4.8. In this example, the projection is trivial for interedges (a, e) and (d, g), that are respectively mapped to part 1 and 2. But, it is clearly ambiguous for vertex f, that is shared by interedges (b, f) and (c, f), since b and c are already mapped to different parts. As the edgecut criterion in G'_{AB} gives the same result for this vertex, one chooses randomly to assign f in part 1.

To implement this operation, we just perform a graph partitioning of G'_{AB} in k parts, where the vertex weight of G'_{AB} is set to zero such that all vertices of G'_A are fixed to their own part according to P'_A, with a vertex weight set to zero. As an optimization, one collapses all vertices of G'_A into k fixed super-vertices that represent each part. By minimizing the edgecut of such an enriched graph, the partitioning routine will compute the desired projection as a balanced partition $\tilde{P}'_B$, assuming P'_A is initially well-balanced.

Note that the repartition, projection and restriction operators use initially fixed vertices in order to model additional constraints on the graph structures. For that, we implement the above operators with a graph partitioning algorithm that supports fixed vertices, as for instance RBBGM or KGGGP as explained in Chapter 3.

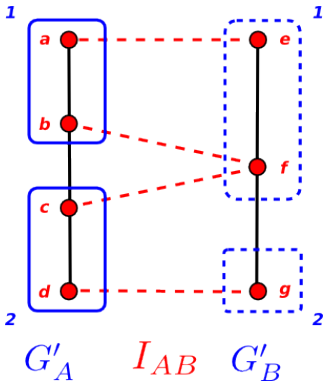


Figure 4.8: Illustration of the projection operator from graph G'_A to G'_B. Given the k'_A-way partition P'_A (in blue), one aims to find a k'_A-way partition of G'_B (in dashed blue), with $k'_A = 2$.

4.3.3 Co-Partitioning Algorithms

We will now present our co-partitioning algorithms, called AWARE, PROJREPART and PROJSUBPART, and the NAIVE method used as the state-of-the-art solution. All these algorithms are precisely described as a sequence of the graph operators previously defined.

For these algorithms, we use as an input the graphs G_A and G_B, the interedges I_{AB} (or $I_{AB}^{-1} = I_{BA}$) and the number of processors k_A and k_B used for both components. Besides, as we do not address the problem of resource distribution (introduced in Section 4.1), we assume that the number of processors used in the coupling phase for both components (i.e., k'_A and k'_B) are also provided as an input). Nevertheless, in section 4.4, we will give a simple heuristic to compute these inputs. All our algorithms solve the co-partitioning problem (defined in Section. 4.3.1) and compute the output partitions P_A and P_B. Let us now give some explanations about our three co-partitioning algorithms:

Naive. As described in Figure 4.9, the Naive method just computes a partition for each graph (G_A and G_B) independently, without taking into account the interedges, as it is currently done in complex coupled simulations. This simple algorithm is clearly not aware of the coupling phase. It will be used as a standard to compare against other methods in the experimental study (Sec. 4.4). In this method, the parameters k'_A and k'_B are not managed, and the coupled partitions P'_A and P'_B are not controlled.

Inputs: G_A, G_B, k_A, k_B
Outputs: P_A, P_B

i. $Part(G_A, k_A) \rightarrow P_A$

ii. $Part(G_B, k_B) \rightarrow P_B$

Figure 4.9: Description of the Naive co-partitioning algorithm.

Aware. This method is divided in three main steps, symmetrically applied to graphs G_A and G_B (Fig. 4.10). It starts by computing the coupled subgraphs (G'_A and G'_B) based on the interedges, using the restriction operator. Then, one partitions each subgraph independently into k'_A and k'_B parts respectively. Finally, the obtained partitions are stretched to the global graphs using the extension operator.

ProjRepart. This method works as the Aware method, except it replaces the "naive" partition of G'_B by the projection & repartition steps (Fig. 4.11). Here, the key idea is to improve the communication scheme during the coupling phase, by minimizing both the maximum volume of communications per processor ($maxV$) and the number of messages ($totZ$). First, the projection operator tries to keep the parts of P'_A face-to-face with those of $\tilde{P}'_B$ (Fig. 4.13a and 4.13b). Then, the repartitioning operator computes the partition P'_B in a

Inputs: $G_A, G_B, I_{AB}, k_A, k_B, k'_A, k'_B$
Outputs: P_A, P_B

i. $Rest(G_A, Dom(I_{AB})) \rightarrow G'_A$

ii. $Rest(G_B, Dom(I_{AB}^{-1})) \rightarrow G'_B$

iii. $Part(G'_A, k'_A) \rightarrow P'_A$

iv. $Part(G'_B, k'_B) \rightarrow P'_B$

v. $Ext(G_A, G'_A, P'_A, k'_A, k_A) \rightarrow P_A$

vi. $Ext(G_B, G'_B, P'_B, k'_B, k_B) \rightarrow P_B$

Figure 4.10: Description of the AWARE co-partitioning algorithm.

way to maintain $maxV$ and $totZ$ quite low (Fig. 4.13c). In practice, it injects the newer parts (in the case where $k'_A < k'_B$) and tries to keep the former ones in place as far as it is possible (minimization of migration volume). Finally, we extend the partitions obtained for G'_A and G'_B to the global graph as we do also in AWARE.

Inputs: $G_A, G_B, I_{AB}, k_A, k_B, k'_A, k'_B$
Outputs: P_A, P_B

i. $Rest(G_A, Dom(I_{AB})) \rightarrow G'_A$

ii. $Rest(G_B, Dom(I_{AB}^{-1})) \rightarrow G'_B$

iii. $Part(G'_A, k'_A) \rightarrow P'_A$

iv. $Proj(G'_A, G'_B, I_{AB}, P'_A, k'_A) \rightarrow \tilde{P}'_B$

v. $Repart(G'_B, \tilde{P}'_B, k'_A, k'_B) \rightarrow P'_B$

vi. $Ext(G_A, G'_A, P'_A, k'_A, k_A) \rightarrow P_A$

vii. $Ext(G_B, G'_B, P'_B, k'_B, k_B) \rightarrow P_B$

Figure 4.11: Description of the PROJREPART co-partitioning algorithm.

PROJSUBPART. This method is a variant of the PROJREPART method in the particular case, where k'_A and k'_B are multiples. One simply replaces the

repartition operator by the *subpartition* operator to compute P'_B from $\tilde{P}'_B$. Other steps are not changed (Fig. 4.12).

Inputs: $G_A, G_B, I_{AB}, k_A, k_B, k'_A, k'_B$
Outputs: P_A, P_B

i. $Rest(G_A, Dom(I_{AB})) \rightarrow G'_A$

ii. $Rest(G_B, Dom(I_{AB}^{-1})) \rightarrow G'_B$

iii. $Part(G'_A, k'_A) \rightarrow P'_A$

iv. $Proj(G'_A, G'_B, I_{AB}, P'_A, k'_A) \rightarrow \tilde{P}'_B$

v. $Subpart(G'_B, \tilde{P}'_B, k'_A, k'_B) \rightarrow P'_B$

vi. $Ext(G_A, G'_A, P'_A, k'_A, k_A) \rightarrow P_A$

vii. $Ext(G_B, G'_B, P'_B, k'_B, k_B) \rightarrow P_B$

Figure 4.12: Description of the ProjSubpart co-partitioning algorithm.

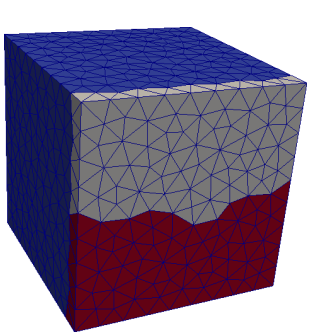
4.4 Experiments

In this section we present the experimental results performed on two sets of data; synthetically generated and real-life coupled simulations.

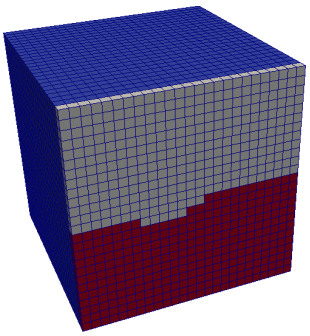
4.4.1 Results on Synthetically Generated Meshes

In this section, we present some experimental results on a synthetically generated data set for the co-partitioning problem. Remember that Naive is our implementation of the most commonly employed method for partitioning a coupled simulation used in industrial and research context. More precisely, Naive partitions each component as an independent problem with any partitioning algorithm (for instance Metis or Scotch). In the remainder of this study, we consider Naive as the reference approach and we compare its quality and performance results to our co-partitioning methods: Aware, ProjRepart and ProjSubpart. A small analysis over the use of a multi-constraint algorithm as a solution to the co-partitioning problem is also included.

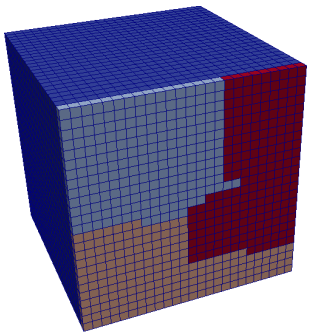
As explained before, we model the repartition, projection and extension operators using initially fixed vertices, so that we need to implement these steps with a partitioning tool that supports such structures. Therefore, in these experiments we consider Scotch, kgggp and rbbgm (presented in Chapter 3) as possible partitioning options. Remember that the above partitioning



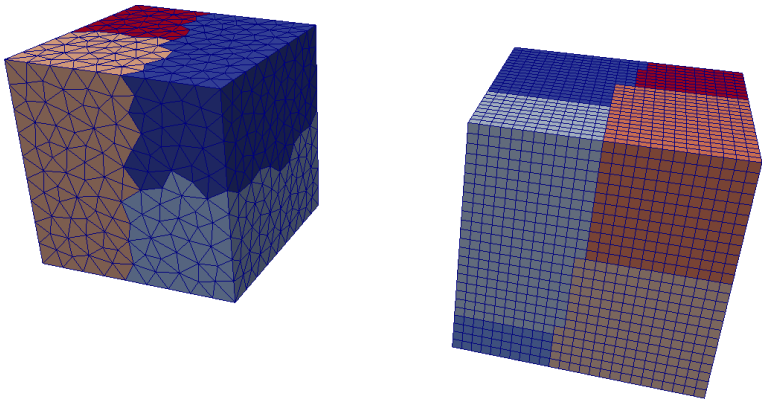
(a) Partition of $G_A^{'}$ in $k_A^{'} = 2$.



(b) Projection to $G_B^{'}$.



(c) Repartition of $G_B^{'}$ in $k_B^{'} = 3$.



(d) Overview of partitions for G_A (left) and G_B (right) after extension ($k_A = 4$ and $k_B = 6$).

Figure 4.13: Example of the *PROJREPART* co-partitioning for a test case similar to `setup3` (see Sec. 4.4).

algorithms are all implemented inside the same multilevel framework and have exactly the same partitioning parameters (coarsening, refinement, imbalance factor, etc).

In our experiments, we also include a version of the multi-constraint algorithm presented in [38], as an alternative solution to the co-partitioning problems. As said before, this method is used for a similar problem, the multi-phase graph partitioning and addresses simulations where computations of different phases have different load characteristics. Here, we implement the MULTICONST method that applies a multi-constraint partitioning for each component separately. Even though the co-partitioning and the multi-phase problem are not equal, in this work, we include results of the MULTICONST approach since it is still an interesting alternative.

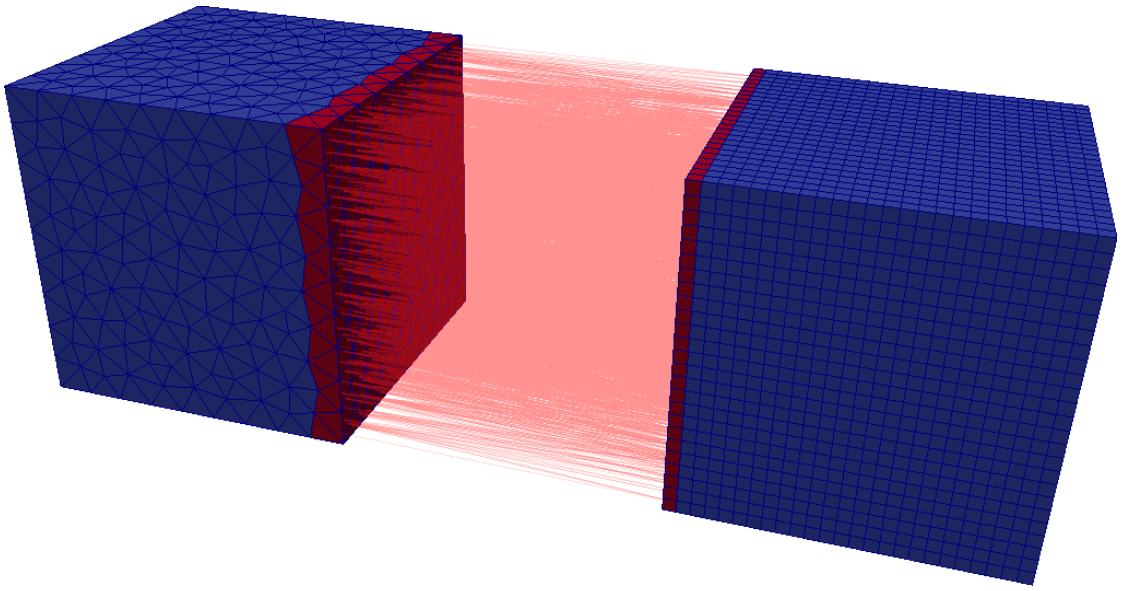


Figure 4.14: Overview of the surface coupling between two mesh domains A (left) and B (right) showing the restriction to the coupled subdomains (in red) and interedges between cells. It corresponds to the `setup3` test case described in section 4.4.

In the following experiments, we use synthetically generated graphs that correspond to mesh structures, and their characteristics are described in Table 4.1. The coupling that we perform in these experiments is a simple surface coupling between two components A and B with cubic domains, and hexahedral or tetrahedral mesh discretization as explained in Figure 4.14. Besides, we assume that all graphs have vertex/edge weigth of 1. In the remainder of this section, we present the results of four experimental setups, named `setup1`, `setup2`, `setup3` and `setup4`, as illustrated in Table 4.2. Finally, we repeat each experiment 5 times in order to compute average values for our results.

Table 4.1: Description of the meshes used in the experiments.

Graph	Elements	$\|V\|$	$\|E\|$
hexa-25x25x25	hexa	15,625	45,000
hexa-70x70x70	hexa	343,000	1,014,300
hexa-100x100x100	hexa	1,000,000	2,970,000
tetra-40630	tetra	40,630	78,131
tetra-928984	tetra	928,984	1,822,323

Table 4.2: Description of the experiments.

Exp.	Graph A	Graph B
setup1	hexa-25x25x25	hexa-100x100x100
setup2	hexa-25x25x25	hexa-70x70x70
setup3	tetra-40630	hexa-100x100x100
setup4	tetra-40630	tetra-928984

To measure the overall partitioning quality, we initially compare the obtained results against the main objectives of partitioning, that is, the global edgecut and partition imbalance for graphs G_A and G_B. In order to measure the co-partitioning quality, we introduce here some additional metrics that give us an insight on the quality of the partitioning during the coupling phase. Thus, an obvious yet important metric in this context is the partition imbalance of the coupled subgraphs G'_A and G'_B. Finally, we are interested in measuring the inter-component communication, so we also compare our results in terms of the number of messages exchanged between the components ($totZ$) and the maximum volume of communication per processor in the coupling phase ($maxV$). Note that, we divide the number of messages exchanged between components into two different types, the major ($totZM$) and minor ones. The minor messages correspond to messages with size equal or less than 5% of the largest message and represent the latency in the network.

First Experiment

In the first experiment, we evaluate the behavior of co-partitioning methods as the number of processors assigned to G_B (k_B) increases. Therefore, we keep the number of processors assigned to G_A (k_A) fixed to a value (16) and for k_B we choose values that are multiples of k_A. Note that an important aspect of our co-partitioning algorithms is that they control the number of processors that are assigned to the coupling phase for both components, k'_A and k'_B. Therefore, for each pair of (k_A,k_B), we find a pair (k'_A,k'_B) where again k'_B is a multiple of k'_A. This is not required by our methods, except for ProjSubpart which is only applicable when k'_B is a multiple of k'_A. Finally, in this first experiment, we include results for different partitioning tools that handle fixed vertices,

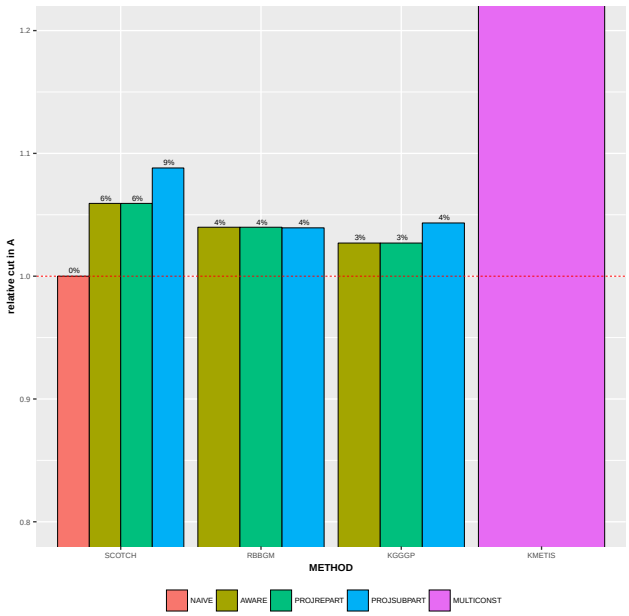
that is SCOTCH, KGGGP and RBBGM (see Chapter 3).

In Figures 4.15a and 4.15b, we present the average edgecut results over all four setups for G_A and G_B respectively. The results are normalized to the values obtained by the NAIVE method with SCOTCH as the partitioning routine. Note that the edgecut results are quite similar among the different setups, so we do not include details on the behavior of our methods for each setup individually. In Figure 4.15a, we see that the edgecut of G_A for AWARE, PROJREPART and PROJSUBPART is not significantly increased compared to the results of the NAIVE approach (less than 5% in average). This increase is conditioned by the partitioning tool that implements the graph operators, and here we see that KGGGP has a minimum increase of 3% followed by RBBGM and SCOTCH with an increase of 4% and 9% respectively. In Figure 4.15b, we notice a relative increase on the edgecut results of G_B for all methods compared to those of G_A. This is mainly due to the fact that our co-partitioning methods are *asymmetric* and impose more constrains on the partitioning of G_B through the use of different operators.

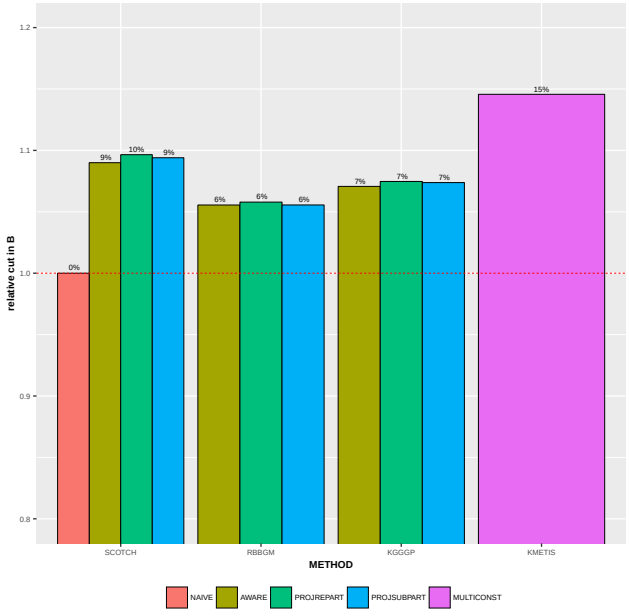
Concerning the internal partitioning routine, one may see that here, RBBGM and KGGGP are better than RB with RBBGM being slightly better than KGGGP (for the chosen graphs). Since the edgecut of G_B is more critical, in the remainder of our experiments, we prefer to use the RBBGM method for the implementation of the graph operators for all co-partitioning methods. (NAIVE still uses SCOTCH since SCOTCH is the best partitioning solution for problems without fixed vertices). Finally, we observe that the MULTICONST method has an edgecut overhead of 23% for G_A and 15% for G_B.

Additionally in Figures 4.16a and 4.16b, we present the imbalance results during the coupling phase for both graphs, G'_A and G'_B. Remember that we measure the imbalance explicitly for the coupling phase following the definition of the co-partitioning and we allow a tolerance of 5% on the average part weight. As one may see, all methods respect the balance constraint for both graphs, except NAIVE, whose imbalance may reach in average 44% for G'_A and 50% for G'_B (not shown in the figures). Remember that we should always make sure that the imbalance constraint during the regular phase is equally respected for both graphs. In this experiment, this is true for all methods except MULTICONST. Indeed, MULTICONST fails to respect the classic load balancing constraint in 31% of the cases. Even though, in this experiment, it slightly overpasses the imbalance tolerance for G'_B or G'_A, this observation is very important and reveals the main limitation of this method. More precisely, MULTICONST does not control the number of processors during the coupling and, as a result, assigns almost all of the available processors of each component to their coupling interface. This problem becomes highly important when the number of processors for each component increases and leads to increased inter-component communications.

Finally, in Figure 4.17, one may see the total number of messages exchanged



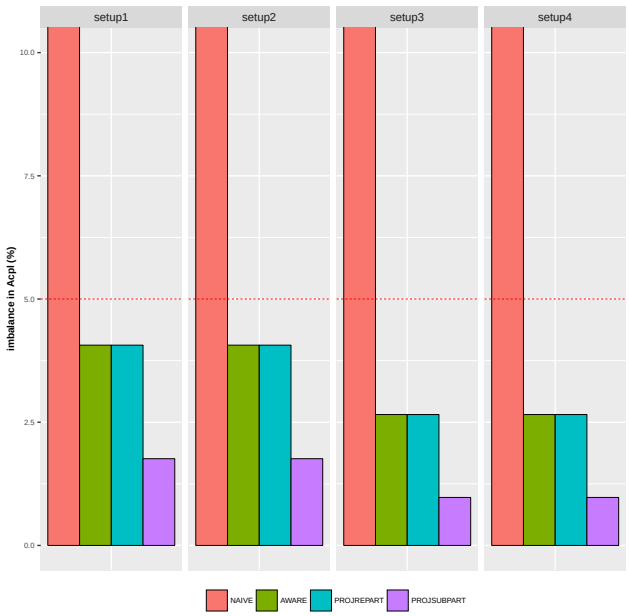
(a) Relative edgecut in G_A.



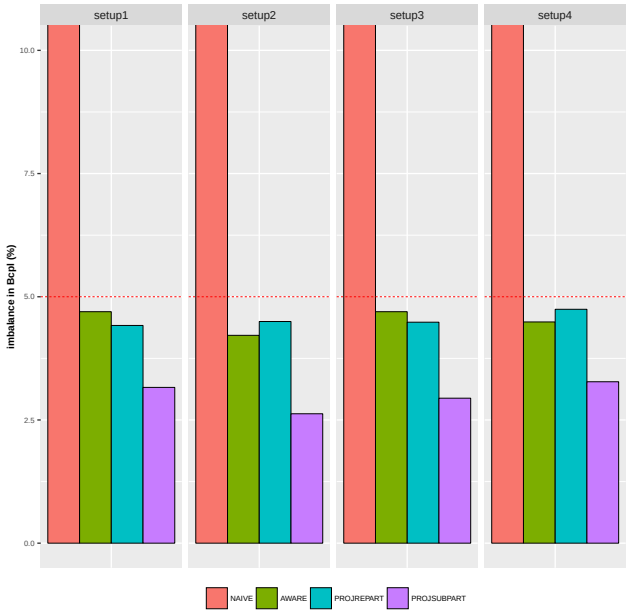
(b) Relative edgecut in G_B.

Figure 4.15: Comparison of co-partitioning algorithms and partitioning methods: relative edgecut mean over all setups.

between the two components ($totZ$) for Aware, ProjRepart and ProjSubpart. We present these results for each setup separately. Note that here we distinguish the number of major and minor messages, coloring the latter with



(a) Imbalance in G'_A.



(b) Imbalance in G'_B.

Figure 4.16: Comparison of co-partitioning algorithms with `RBBGM` method: mean imbalance in coupling phase, over all setups (imbalance tolerance fixed to 5%).

a yellow shade. Additionally, the dashed red line represents the optimal number of messages that may be exchanged during the coupling phase among the

processors of the two components. The calculation of the optimal $totZ$ follows the formula: $totZ \geq k_A^{'} + k_B^{'} - \gcd(k_A^{'}, k_B^{'})$, presented in [69].

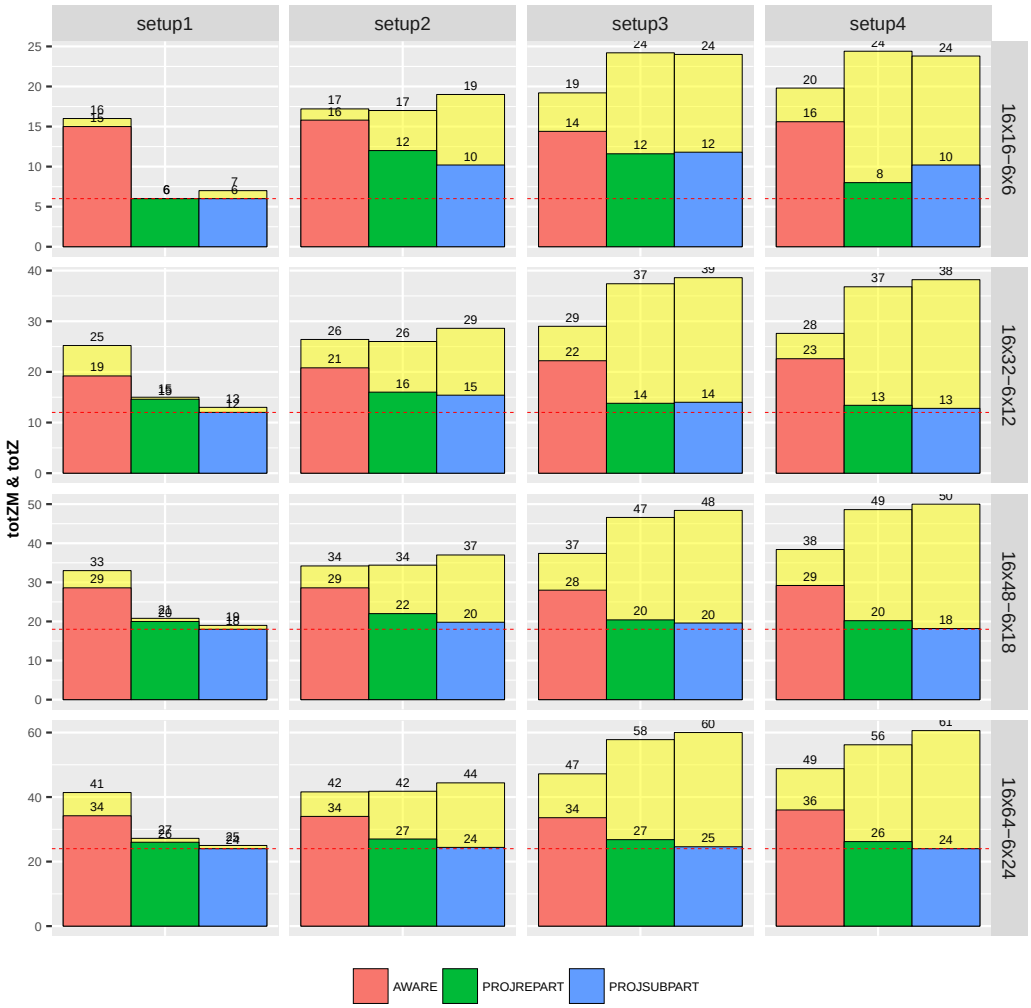
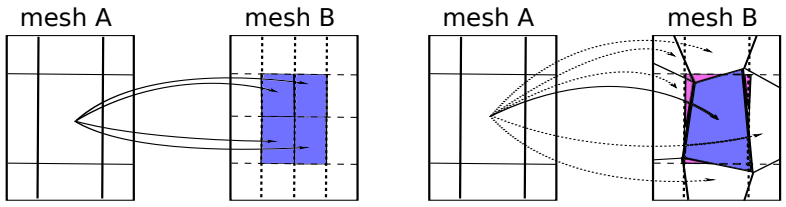


Figure 4.17: Comparison of co-partitioning algorithms with `RBBGM` method: total number of major messages $totZM$ and total number of messages $totZ$ for each setup separately.

From the above formula, it is easy to see that the total number of messages depends on the number of active processors in the coupling interfaces of both components, $k_A^{'}$ and $k_B^{'}$. In this experiment, we compare the results of our co-partitioning methods to the optimal number of messages and not to the results of Naive. We do that since the number of processors during the coupling phase may be different between Naive and our co-partitioning methods. Remember that Naive does not control $k_A^{'}$ or $k_B^{'}$, so these values may be different and would lead to an unfair comparison with other methods where $k_A^{'}$ and $k_B^{'}$ are

imposed. More precisely, NAIVE assigns almost the same number of processors for $k_A^{'}$ compared to AWARE, PROJREPART and PROJSUBPART, but $k_B^{'}$ highly differs from that of AWARE, PROJREPART and PROJSUBPART as the total number of processors for component B increases and does not allow us to draw clear conclusions. Results for the MULTICONST are also omitted for the same reasons. Remember that MULTICONST assigns many processors in the coupling phase leading to a large number of messages during coupling.

Due to the differences in the mesh discretization between component A and B, the problem of minimizing the communication costs in the coupling interface is more complex for the setups `setup2`, `setup3` and `setup4` than for `setup1`. Indeed, for `setup1` we use similar mesh discretization (with hexaedral elements well aligned), unlike for `setup2`, `setup3` or `setup4`, where we use misaligned elements (hexaedral or tetraedral). In Figure 4.18, we illustrate two examples of mesh alignment in the coupling interface along with the messages to be exchanged in each case. In 4.18a, there is an exact correspondence of one element in mesh A to several elements in mesh B, like for `setup1`. On the contrary, in 4.18b, a bad alignment between the two meshes creates possible additional messages, denoted with the dashed lines, like for `setup2`, `setup3` and `setup4`. These latter messages correspond to minor messages that, as said before, may represent the latency in a network. Therefore, in Figure 4.17, we see that for `setup1`, where the mesh structures of A and B are well aligned, the number of messages is minimized for both PROJREPART and PROJSUBPART, but not for AWARE. This is expected since AWARE does not minimize the number of messages and does not take into consideration the intersection of different mesh elements on the coupling interfaces (as do PROJREPART and PROJSUBPART). For the other setups, we see that PROJREPART and PROJSUBPART do not minimize the total number of messages exchanged between components ($totZ$), but they do minimize the number of major messages, which is often equal to the optimal value. Note, that is an interesting result since, under a fast network the minor messages may be ignored.



(a) Good mesh alignment. (b) Bad mesh alignment.

Figure 4.18: Examples of different mesh alignment in the coupling interface.

Additionally, in Figure 4.19, we present results on the maximum volume of inter-component communications ($maxV$) for each setup separately. By

definition, we calculate $maxV$ for the component with the smaller number of processors on the coupling interface, since the maximum volume per processor is greater when the number of processors is smaller. Remember that we take $k_A^{'} \leq k_B^{'}$, so we calculate the $maxV$ for component A. In this experiment, for AWARE, PROJREPART and PROJSUBPART $k_A^{'}$ is set to 6, while the number of processors that are active in the coupling interface of component A for NAIVE is between 6 and 8. Considering that $k_A^{'}$ is almost equal for all methods, we may include the results of NAIVE in the comparison of $maxV$ (with a slight advantage for NAIVE).

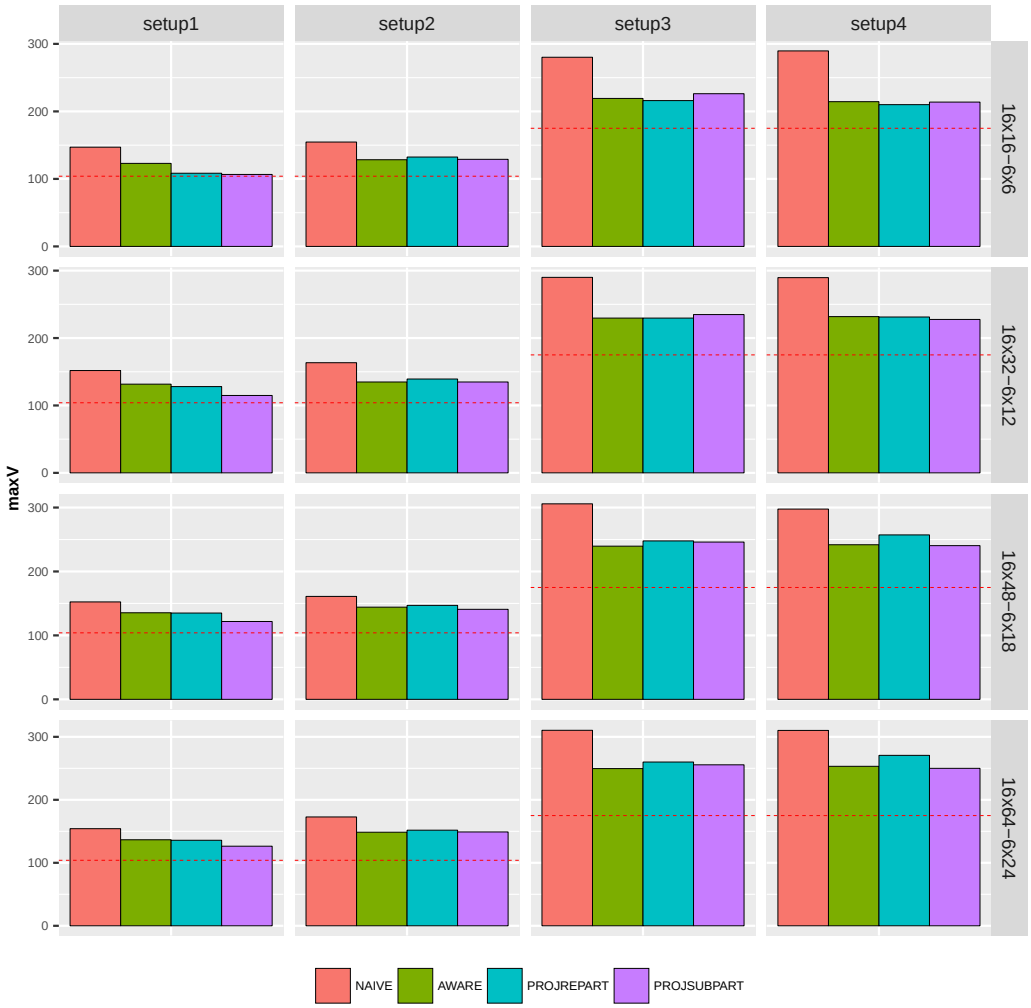


Figure 4.19: Comparison of co-partitioning algorithms with `RBBGM` method: maximum volume of inter-components communication for each setup separately.

As a result in Figure 4.19, we may confirm that our co-partitioning al-

gorithms manage to minimize the inter-component communication costs of the coupling phase, since $maxV$ is minimized compared to the results of the NAIVE method (up to 26%). To conclude the analysis of this experiment, in 4.20, we present the average execution time for each co-partitioning algorithm relative to the NAIVE method. As one may observe the relative results of AWARE, PROJREPART and PROJSUBPART are, in the worst case, 16% slower that NAIVE which is expected to be faster since it is a more straightforward solution.

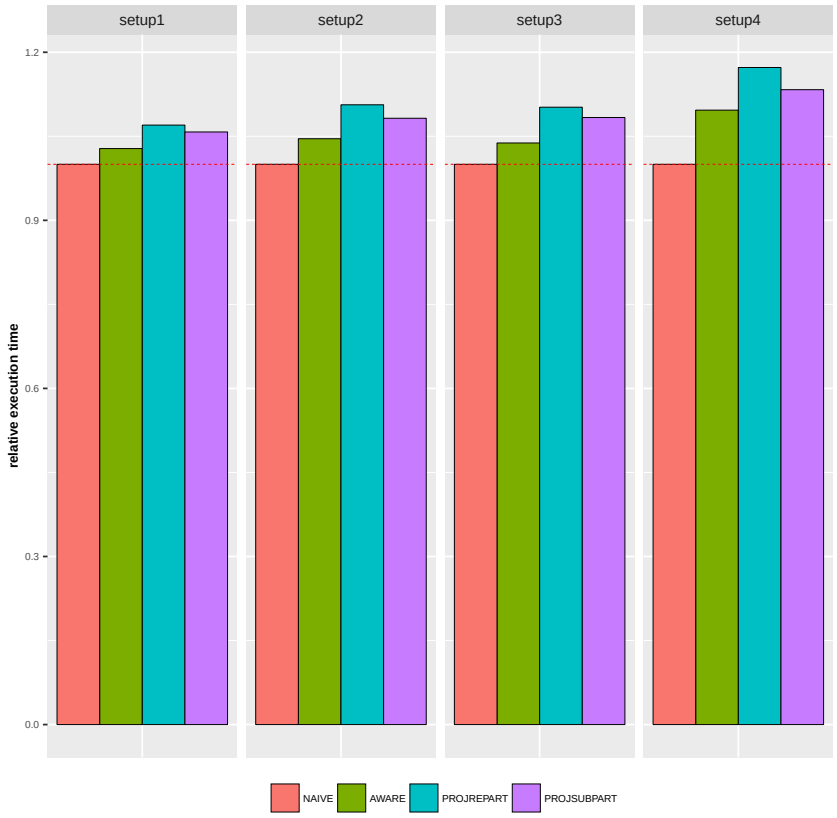


Figure 4.20: Comparison of co-partitioning algorithms with `RBBGM` method: relative time, mean over all setups.

Second Experiment

The second experiment that we present here studies the behavior of the co-partitioning algorithms as the number of processors in the coupling interfaces for component B (k'_B) increases. In order to perform this experiment, we focus on one setup case (`setup4`) and we fix the number of processors for G_A and G_B, such that $k_A = 16$ and $k_B = 48$. Additionally, we also fix k'_A to 6 and we vary the values of k'_B in a range of $[k'_A, k_B]$ (with a step of two). Since in this

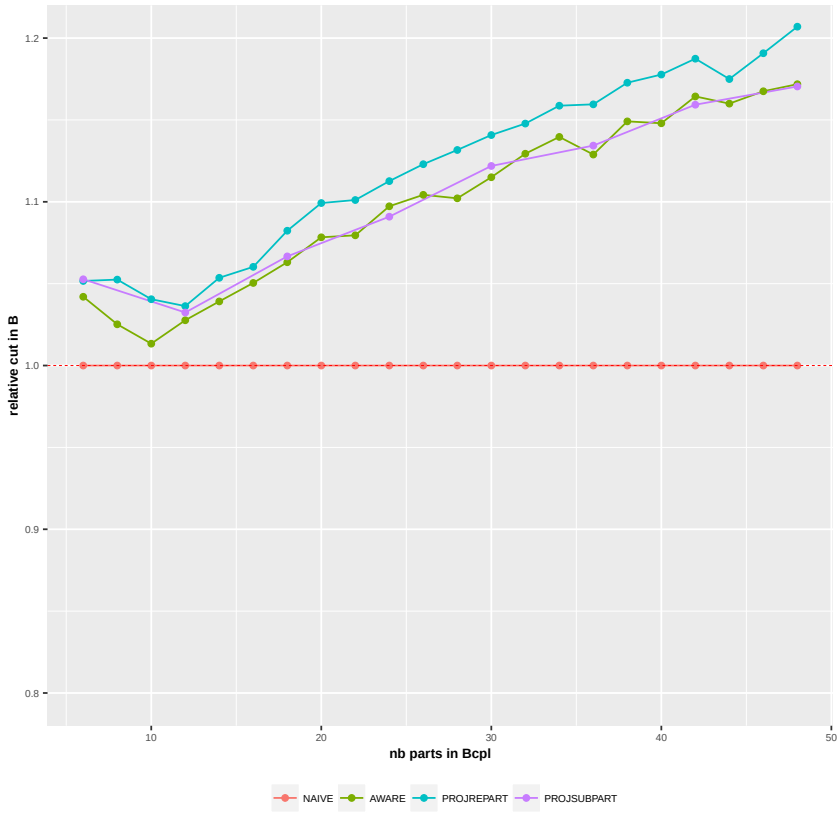


Figure 4.21: Comparison of co-partitioning algorithms with `RBBGM` method: edgecut of G_B as $k_B^{'}$ increases. Results for `setup4` when $k_A = 16, k_A^{'} = 6, k_B = 48$.

experiment we test the behavior of our co-partitioning methods for the most critical component B, results on component A (edgecut and load imbalance) are omitted but they follow the analysis of the first experiment.

In Figure 4.21, we present the edgecut results of G_B for our co-partitioning methods normalized to the results of Naive. Note that since ProjSubpart may be applied only when $k_B^{'}$ is a multiple value of $k_A^{'}$, consequently some results are missing for this method. In this experiment, we observe that the edgecut results of G_B increase for all methods as the number of processors in the coupling phase increases.

The method with the best relative edgecut results is Aware and the one with the worst results is ProjRepart, which is expected if we consider the additional constraints of ProjRepart compared to Aware. Note that the edgecut increases for the Aware method in the range of 1% to 17% compared to Naive, while for ProjRepart is on average 4% more than Aware. It is interesting to see that the best edgecut results of all methods appear when the number of processors in the coupling phase is close to the the geometric ratio

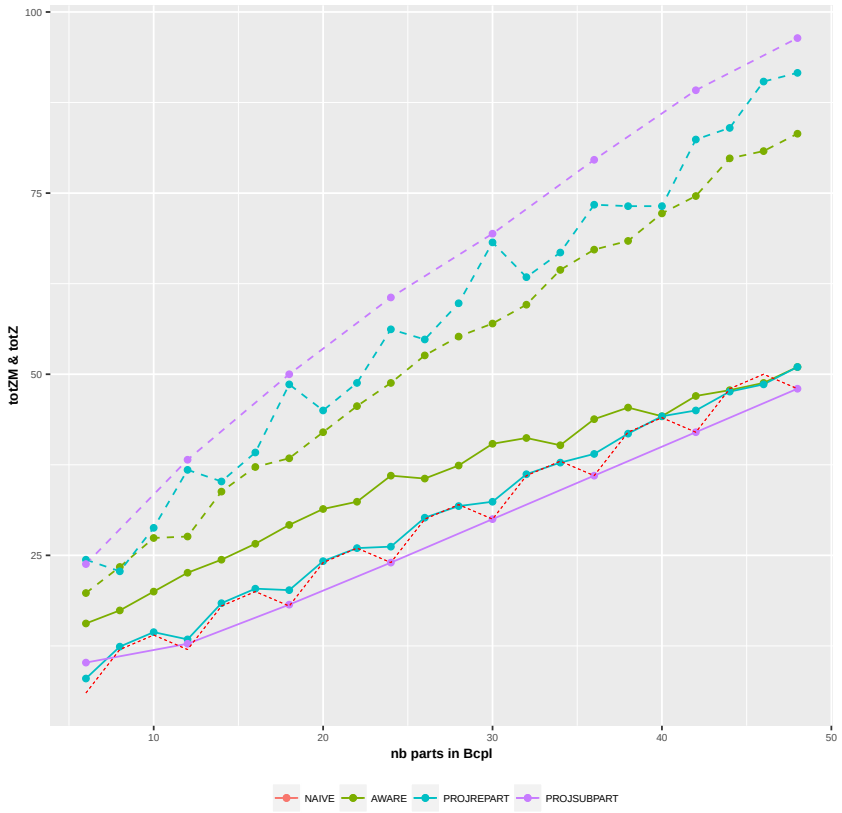


Figure 4.22: Comparison of co-partitioning algorithms with `RBBGM` method: $totZ$,$totZM$ as k'_B increases. Results for `setup4` when $k_A = 16, k'_A = 6, k_B = 48$.

between the coupling surface and the entire cubic domain, i.e., $k'_X = \left\lfloor {k_X}^{2/3} \right\rfloor$ and here $k'_B = 48^{2/3} = 13.2$.

Again for this experiment, all methods except Naive manage to balance the additional constraint of load balance in the coupling phase for both components A and B, that explain this edgecut overhead. On the other hand, Naive has a huge imbalance factor of 68% for G'_B.

In Figure 4.22, we present the total number of messages, major and minor, exchanged during the coupling phase for this experiment. The total number of messages ($totZ$) for each method is represented with dashed lines, while the number of major messages ($totZM$) is represented with plain ones. Here, we compare our results with the optimal number of messages represented with a red dashed line. First, we observe that the total number of messages increases linearly with the increase of k'_B for all methods, which is expected. Additionally, we see that both ProjRepart and ProjSubpart minimize the number of major massages compared to Aware, when k'_B remains relatively small.

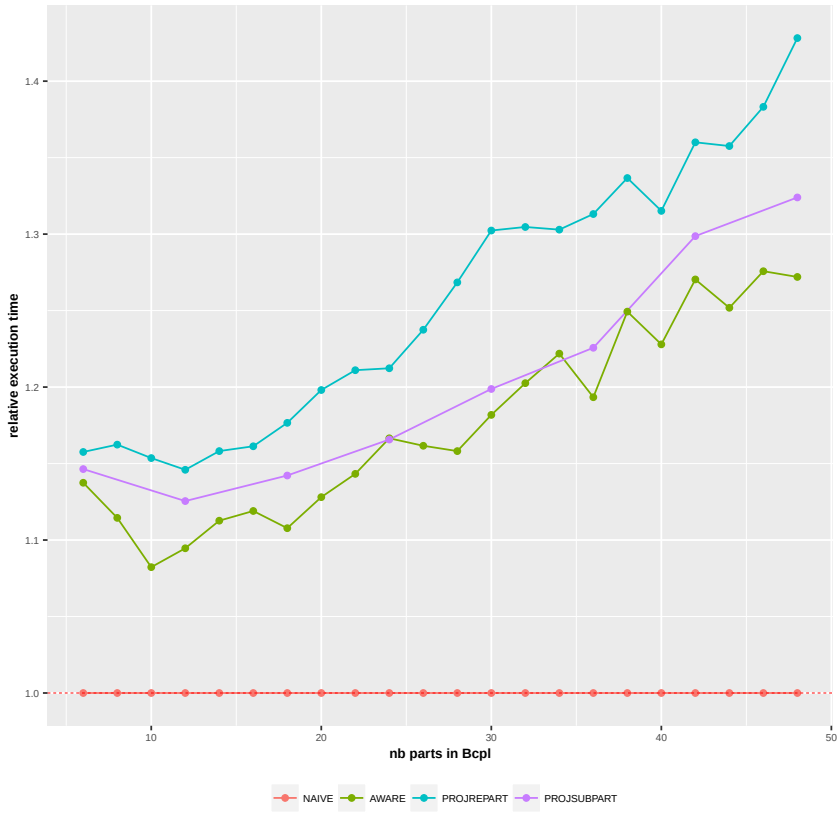


Figure 4.23: Comparison of co-partitioning algorithms with `RBBGM` method: relative time as $k_B^{'}$ increases. Results for `setup4` when $k_A = 16, k_A^{'} = 6, k_B = 48$.

However, since in this setup (`setup4`) the mesh elements between the two components are not well aligned, there is a large number of minor messages. As said before, under a fast network, those minor messages may be ignored as they represent negligible volume ($< 5\%$) compared to major messages.

Finally, in Figure 4.23, we present results for the runtime performance of the co-partitioning algorithms compared to Naive. We observe that our algorithms are getting slower as $k_B^{'}$ increases since the problem becomes more complicated regarding the additional constraints. Note that since Aware has less constraints to optimize, it is in general faster that the other two methods.

Third Experiment

The final experiment that we present here aims to determine the quality of the co-partitioning methods as the coupling interfaces increase, from a surface overlap to a fully volumetric one. This experiment is only performed in the `setup2` and the overlap may be in the surface (0% of overlap), at 25% of the whole volume, at 50% and at 100%, full overlap. In the following figures the

overlap is denoted as 0,25,50 and 100 respectively.

In Figure 4.24, we illustrate the edgecut results for G_A and G_B relatively to the NAIVE method for different combinations of k_A, k_B and $k_A^{'}, k_B^{'}$. The number of $k_A^{'}$ and $k_B^{'}$ change proportionally to the volume of overlap. Here, we see that the results are rather good for all methods and as the overlap becomes larger (in volume), we notice that the edgecut minimization increases for both components. We remark that when the coupling interface is small, the co-partitioning algorithm has not enough information on the entire domain. So a good partition of a small surface may lead to a bad partition of the entire domain due to lack of global view in the *extension* operator.

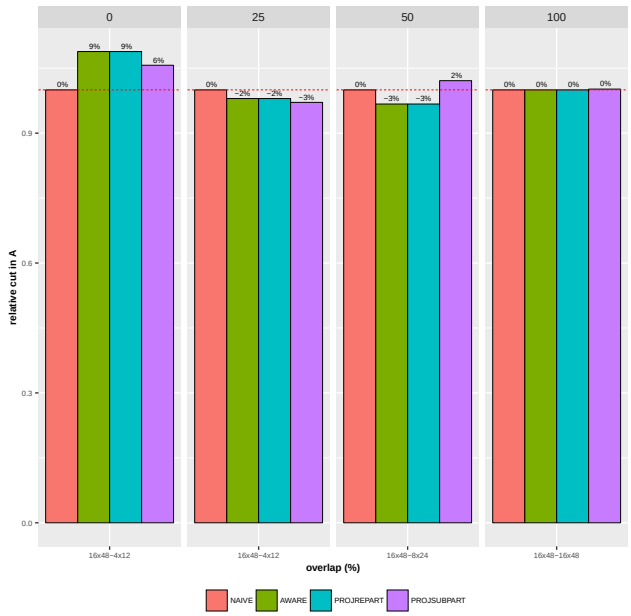
For the load imbalance of $G_A^{'}$ and $G_B^{'}$ during the coupling phase, we note that our co-partitioning algorithms respect this constraint as opposed to NAIVE, but sometimes fail to respect the imbalance factor for the entire graphs G_A and G_B. This leads us to believe that our co-partitioning methods are not yet robust enough to handle larger coupling overlaps, and more care should be taken, due to the large number of interdeges involved in these cases.

Moreover, in Figure 4.25, one may see the number of major and minor messages during coupling as the overlap increases. In this experiment, we confirm the observation from previous experiments that PROJREPART and PROJSUBPART minimize the number of major messages compared to AWARE. Additionally, as the overlap becomes larger, the number of major messages becomes smaller. However, note that there is a large increase of minor messages that correspond to a bad alignment among mesh elements of different types. This is rather expected under a volumetric overlap. Finally, we notice (but do not depict here) that the execution time of PROJREPART and PROJSUBPART becomes many times slower than NAIVE or AWARE as the overlap increases.

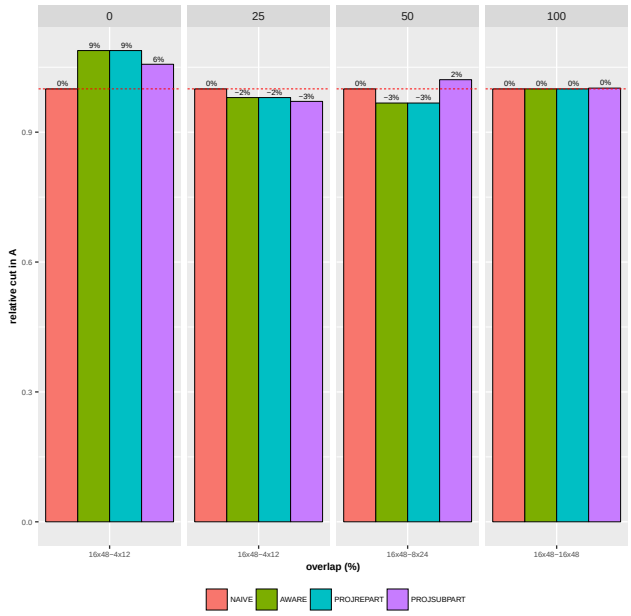
4.4.2 Overview of a Real Application

In this section, we present in detail an example of a coupled simulation (provided by CERFACS) that is used in real industrial configurations in order to optimize the design of gas turbines [34, 36, 65, 74, 75]. Later we will present experimental results for our co-partitioning algorithms on a data set related to the above simulation. Experts in the field of computational physics agree that multi-physics modeling approach is an encouraging alternative in reducing the costs and the duration of the optimization of gas turbines. In this context, an essential part of the problem is to study the temperature field in a turbine blade placed in the high pressure distributor of a helicopter.

More precisely, the configuration of an helicopter combustion chamber is presented in figure 4.26a. The function of the combustion chamber is to provide an enclosure in which large quantities of fuel, supplied through the fuel spray nozzles (B), are burnt with extensive volumes of air provided from the secondary air flow (A). This is achieved by means of a flame tube (C) which



(a) Relative edgecut in G_A.



(b) Relative edgecut in G_B.

Figure 4.24: Comparison of co-partitioning algorithms with `RBBGM` method: edgecut results for `setup2` as the coupling overlap increases from 0 to 100.

contains a high pressure distributor with a blade of the downstream stator (D) that controls the airflow along the chamber. Finally uniformly heated gas is

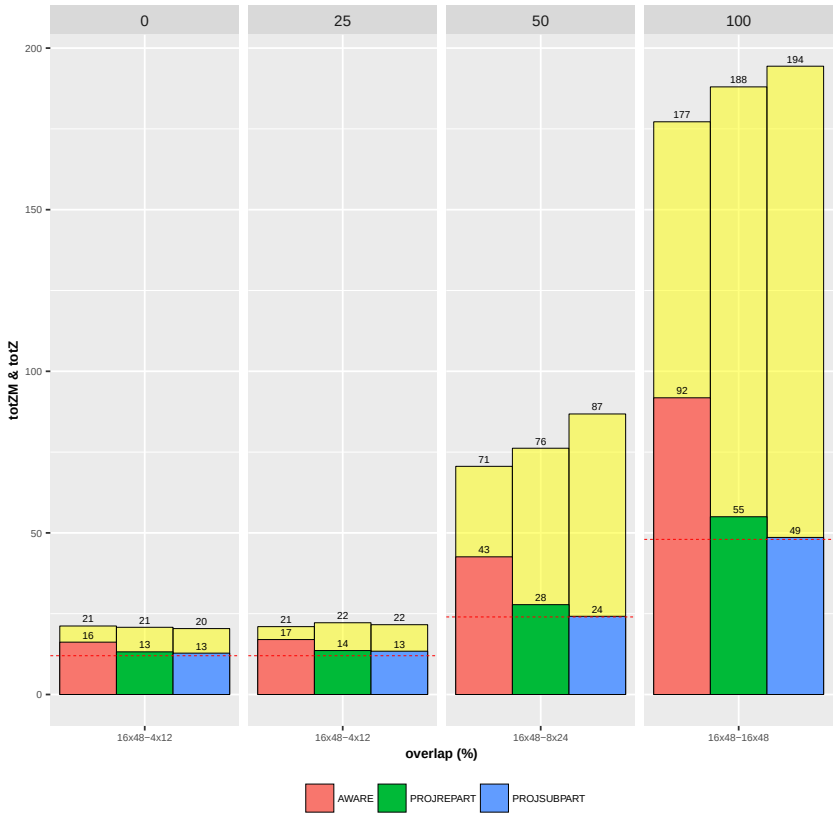
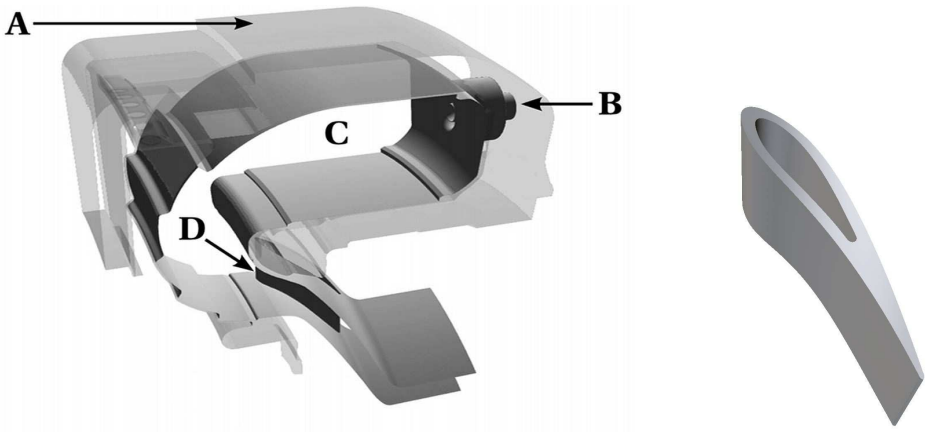


Figure 4.25: Comparison of co-partitioning algorithms with `RBBGM` method: $totZM$, $totZ$ for `setup2` as the coupling overlap increases from 0 to 100. The optimal value for $totZM$ is given by the red dashed line.

released as required by the turbine.

Chamber cooling is achieved by cold air films along the walls. The main thermal problem here is linked to the blade 4.26b, which does not hold the high temperatures produced by the flame in the combustor. In order to decrease their temperature, the burnt gases are therefore mixed with cold air in the dilution zone of the combustion chamber. The challenge is to reach an homogeneous mixture at the chamber exit and guarantee the absence of hot spots that could damage the blade.



(a) Sections of a combustion chamber. (b) Blade.

Figure 4.26: Presentation of geometric configuration inside a combustion chamber.

Recently, researchers use Large Eddy Simulation (LES)[3] in order to model realistic combustors. However, if it is not coupled with other physical phenomena, its applicability remains limited for thermal aspects [10]. For instance, when flames interact with the walls of a chamber, a simultaneous resolution of the temperature may be required within the solid and around it. Therefore, in order to improve the quality of the simulation, additional distinct numerical models that account for radiation and heat transfer should be introduced in the process. In other words, the objective is to evaluate the impact of heat conduction and radiation on the thermal behavior of the blade in the combustion chamber.

More precisely, in this particular study performed by CERFACS, three different solvers that represent distinct physical properties are coupled together to simulate a realistic combustion chamber provided by Turbomeca. The LES in this implementation is performed with the CFD code AVBP which is developed at CERFACS and solves the Navier-Stokes equations together with the energy and chemical species conservation equations. The calculation of thermal diffusion in solids is performed with the code AVTP, which is also

[3] LES is a mathematical model for turbulence simulation, where a low-pass filtering of the Navier–Stokes equations is used to reduce the computational cost.

developed at CERFACS and solves the classic heat equation. Finally, the radiation solver, called PRISSMA, has been specifically designed for combustion applications and uses discrete-ordinates-method (DOM) with different angular discretizations and spectral models. DOM allows the use of the same kind of mesh as used in LES, which is a great advantage for the coupling. The interactions of the three physical phenomena, i.e. combustion, heat conduction and radiation are illustrated in figure 4.27. Couplings occur between AVBP and AVTP, PRISSMA and AVTP and finally PRISSMA and AVBP. Note that the first two are surface couplings and data are exchanged at the boundaries of the flow/solid domain, while the last one is volumetric. More details on the data that are exchanged and the underlying physical interpretation during the coupling can be found in [34]. Therefore, the problem of gas turbine optimization is governed by two distinct coupled simulations, the first one couples the LES with the heat conduction (denoted AA) and involves AVBP and AVTP, while the second one couples the LES with heat conduction and radiation simulation (AAP) and involves all three solvers.

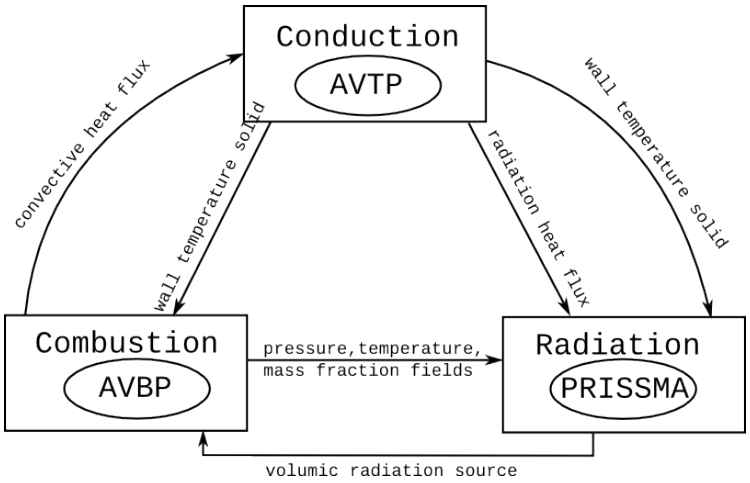


Figure 4.27: Interactions between combustion, radiation and heat conduction in solids.(modified version of figure in [34]).

The strong differences in time scales between the three physical phenomena requires a coupling strategy. More precisely, the coupling between AVBP and AVTP is not synchronized in physical time, since each model has a different characteristic time, implying different time steps. Also the radiation source exchanged between AVBP and PRISSMA must be updated at time intervals corresponding to the characteristic time scale of the temperature and mass fractions, imposing dependencies.

All three solvers used here are efficiently parallelized; they use subdomain decomposition, and in the case of PRISSMA, the calculation is also parallelized using a spectral and angular discretization. However, the three solvers have very different execution times. Typically, AVTP is very fast (0.96

Table 4.3: Processor distribution and CPU time. The quantity $CPU_{0.5}$ is the CPU time needed to compute 0.5ms of physical time.

simulation	AVBP	AVTP	PRISSMA	$CPU_{0.5}$
case AA	230	24	-	1hr
case AAP	105	12	48	3hr

s/iteration/processor), as it involves a simple equation for only one variable, whereas PRISSMA is very long (3000 s/iteration/processor), mainly owing to the spectral complexity and the non-local character of the integration (optical paths go through the whole domain). AVBP is in between, with 95.2 s/iteration/processor, but requires many iterations. As already explained before, a coupling framework is necessary in most coupled simulations as it enables the communication of the components at their coupling interfaces. In this application, the OpenPALM coupler is used which follows a fully distributed, direct communication scheme (DCS).

Table 4.3 gives the processor distribution as well as CPU times for the different solvers in the two coupled simulations (AA and AAP), as extracted from [34]. Note that AVBP and AVTP run on half of the processors in the AAP simulation compared with the AA simulation. Because PRISSMA reaches a speedup limit at 48 processors, load balancing and synchronization imply a limitation of processors on the two other codes.

Researchers working on this problem confirm that an efficient coupling of the above simulations may be achieved with better resource allocation strategy. Moreover, they believe that a solution to the data distribution for a coupled simulations may be an important step towards a high performance. As they state in [36], the time spent during the inter-component synchronizations in AA may reach approximately up to 30% of the total execution time.

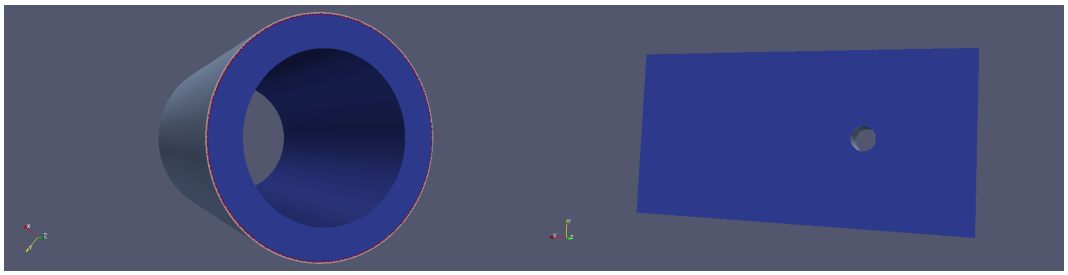
4.4.3 Results on a Real-life Application

Finally, in this section we present experimental results on a test case for the AA coupling simulation between AVBP and AVTP for the problem of gas turbine optimization, as presented in the previous section. The meshes that are used to describe the two components along with some characteristic values can be found in Table 4.4. Additionally, in Figure 4.28, we give an illustrative overview of the two meshes and the coupling interface of this simulation. As one may see in the figures, the coupling interface in this configuration is rather small compared to the domains of both mesh structures. Nevertheless the above example is sufficient considering that this is a first evaluation of the co-partitioning techniques for real applications. For these experiments we choose AVTP to represent component A and AVBP to represent component B.

In order to evaluate the behavior of our co-partitioning methods in a real coupling configuration, we perform here similar experiments as the ones per-

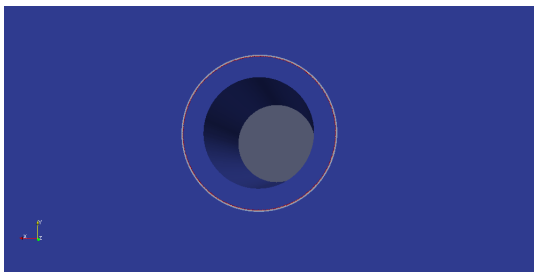
Table 4.4: Information on the graph and mesh structure of the real-life application.

`cylinder`	#vertices	#edges	mesh elements
meshA	38,080	75,406	hexa and prism
meshB	319,492	639,724	prism



(a) Mesh overview of component A.

(b) Mesh overview of component B (the coupling interface is inside the perimeter of the circle).



(c) Coupling intersection.

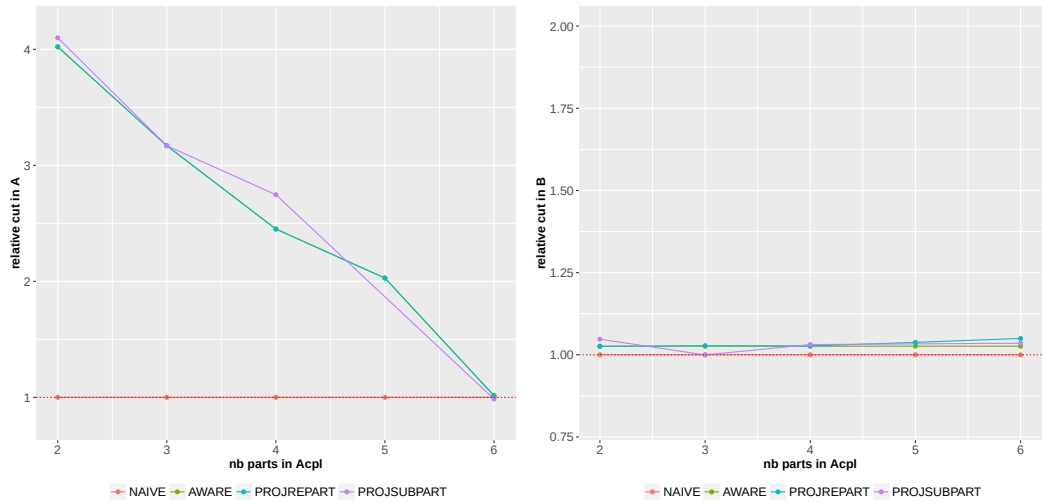
Figure 4.28: Illustration of the mesh structures along with the coupling interfaces of components A and B of `cylinder` test case.

formed for the synthetically generated test cases in section 4.4.1. After these experiments we will be able to determine the best parameters for the real-life coupled simulations and we will have an insight on the limitations of our methods.

First, we study the behavior of our methods as the number of processors assigned in the coupling interface of each component increases. We perform the experiment for both components A and B in order to evaluate the best values of $k_A^{'}$ and $k_B^{'}$ for our co-partitioning methods. To start, we randomly choose $k_A = 6$ and $k_B = 26$. In the first experiment, $k_A^{'}$ takes values from $[2, k_A]$ while $k_B^{'}$ remains fixed. On the other hand, in the second experiment $k_A^{'}$ remains fixed while $k_B^{'}$ increases from $[k_A^{'}, k_B]$. For the first experiment we also evaluate the impact of the projection operator by monitoring the edgecut increase of G_B as $k_A^{'}$ increases.

The edgecut results for the first experiment and for both components A and B are shown in Figure 4.29. More precisely, in Figure 4.29a, we observe that

the edgecut of G_A is highly dependent on the number of processor assigned in the coupling phase and can be many times worse than that of NAIVE. It is interesting to see that the best edgecut result for all co-partitioning methods has a minimal overhead of 1% and is achieved when $k_A^{'} = k_A$. A series of tests on different input values k_A and k_B and varying $k_A^{'}$ confirm the above results but are not shown here, since they follow a similar pattern. Additionally, if we consider the mesh of component A (in Figure 4.28a), we remark that its geometric structure favors a partitioning where $k_A^{'} = k_A$, due to the cylindric form. Finally, in Figure 4.29b we see that the edgecut of component B for PROJREPART and PROJSUBPART remains relatively unaffected as $k_A^{'}$ increases. This indicates that the projection operator between component A and B does not introduce an edgecut overhead whatever the value of $k_A^{'}$. Indeed, the edgecut remains rather constant at a maximum increase of 6% relative to NAIVE.



(a) Relative edgecut in G_A.

(b) Relative edgecut in G_B.

Figure 4.29: Comparison of co-partitioning algorithms with `RBBGM` method: edgecut of G_B and G_A as $k_A^{'}$ increases. Results for `cylinder` when $k_A = 6, k_B = 26$ and $k_B^{'} = 12$.

In Figure 4.30, one may see the edgecut results for component B as the number of $k_B^{'}$ varies while $k_A^{'}$ is fixed to k_A (following the results of the previous experiment). Note that, the edgecut of component B is not heavily affected by the different values of $k_B^{'}$ and the maximum edgecut increase is only 8% relative to NAIVE. More precisely, the worst results are obtained when $k_B^{'} = k_A$, while the best results are obtained when $k_B^{'} = k_A^{'} = 6$. This means that component B is not as critical as component A regarding the partitioning overhead introduced in the co-partitioning.

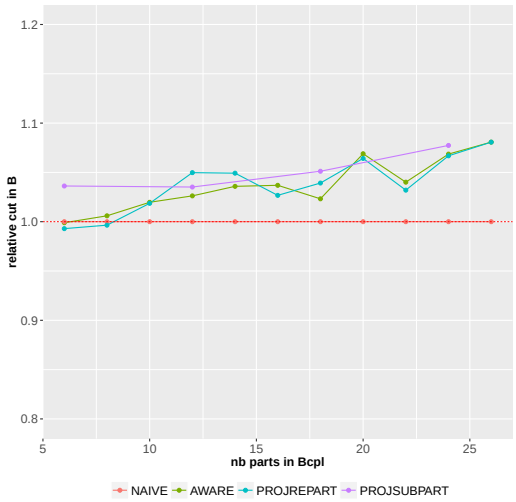
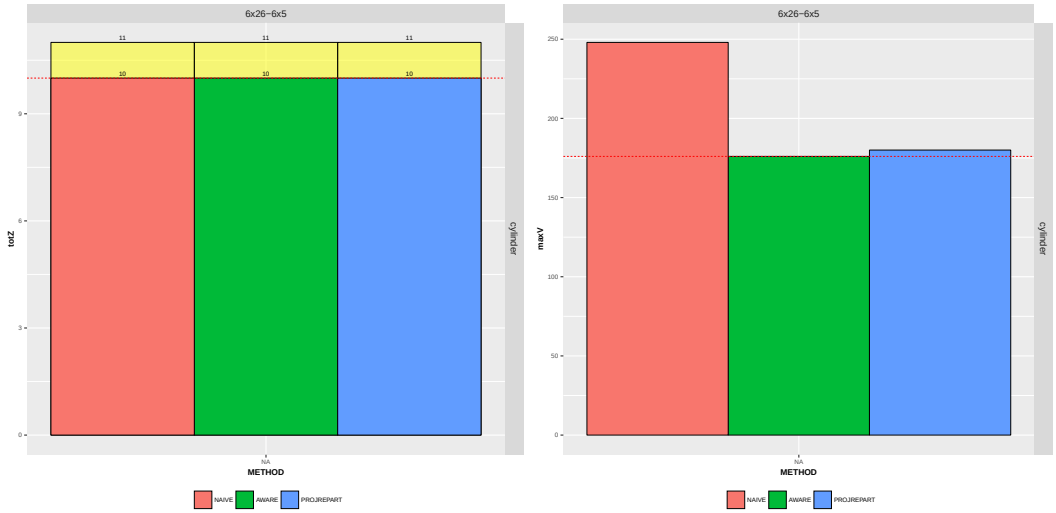


Figure 4.30: Comparison of co-partitioning algorithms with `RBBGM` method: edgecut of G_B and G_A as $k_B^{'}$ increases. Results for `cylinder` with $k_A = 6, k_B = 26$ and $k_B^{'} = 6$.

Moreover, we compare the $totZ$ and $maxV$ of our co-partitioning methods to the results of Naive. Remember that for a fair comparison of the above metrics, all methods should have the same values of $k_A^{'}$ and $k_B^{'}$. Therefore, we set the above values for our co-partitioning methods equal to the values indicated by Naive, that is 6 and 5 respectively. As we may see in Figure 4.31a, all methods (including the Naive) exchange the same number of total messages where the number of major messages is equal to the optimal value for all methods. Additionally in Figure 4.31b one may see that the maximum volume of communication is minimized for Aware and ProjRepart as opposed to Naive that has an increased $maxV$ of 30% (ProjSubpart is not included since $k_B^{'}$ is not a multiple of $k_A^{'}$).

Following, we perform a final experiment with random number of k_A and k_B where $k_A^{'}$ and $k_B^{'}$ are carefully chosen as indicated by the two first experiments. In Figures 4.32, one may see the edgecut results for components A and B respective where the maximum edgecut increase is only 7% compared to Naive. As we mentioned before, an edgecut increase for G_A or G_B is rather expected due to the additional constraints imposed by the co-partitioning problem. Here, we see that if the number of $k_A^{'}$ and $k_B^{'}$ are carefully selected, this increase may be minimised. Finally, in Figures 4.33 we present the imbalance results during the coupling interface for both meshes, where, as expected, our co-partitioning algorithms respect the additional balance constraint, while Naive does not.

To conclude, we present performance results of the `cylinder` case, executed on a the *Neptune* cluster at CERFACS. The cluster includes 2,528 cores

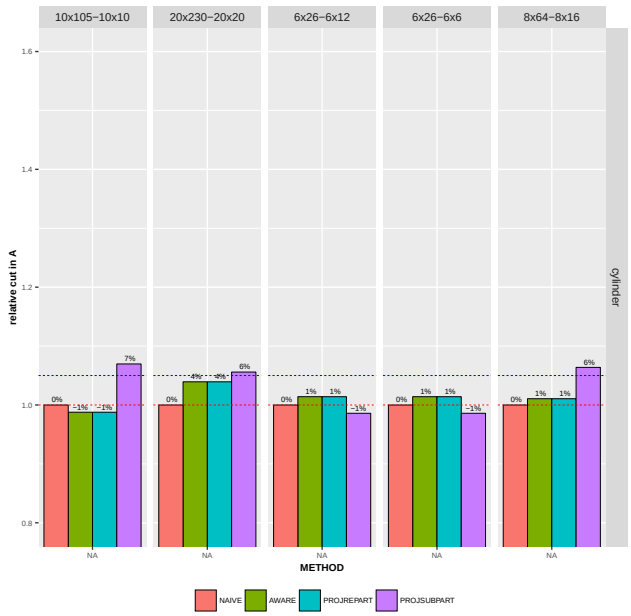


(a) $totZ$ and $totZM$. (b) $maxV$.

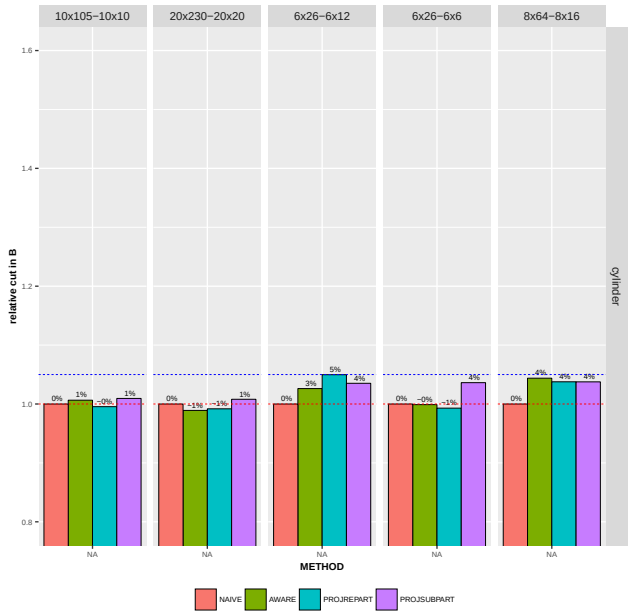
Figure 4.31: Comparison of co-partitioning algorithms with `RBBGM` method: inter-component communication costs for NAIVE, AWARE and PROJREPART for same values of k_A, k_B, k'_A, k'_B. Results for `cylinder`.

distributed in 158 computational nodes each having 2 processors Intel with together 8 cores SandyBridge running at 2.6 GHz with 32 GB of DDR3 memory. The network of the cluster is an Infiniband QDR non blocking network. In these preliminary experiments we compare the PROJREPART method with the NAIVE for the test cases presented in the experiment presented just before (4.32 and 4.33). More precisely, $k_A \times k_B$ take the following values: $10 \times 105, 20 \times 230, 6 \times 26$ and 8×64 respectively and for PROJREPART the additional $k'_A \times k'_B$ parameters are chosen as $10 \times 10, 20 \times 20, 6 \times 6, 8 \times 16$. The partitions computed for PROJREPART are loaded in each solver at runtime. In Table 4.5, we present the average time per iteration of the regular phase taken on the slower component and the average time per coupling iteration for the `cylinder` case. Note that the time spent in the regular phase is not degraded when the PROJREPART is used, which reflects the minimum edgecut increase presented in 4.32. As one may see here, the results indicate that a minimization of the coupling time may be achieved but further investigation should be made in order to analyze the performance of the co-partitioning algorithms.

Since in this experiment, k_A and k_B are rather small the overall coupling time is not significant and reaches up to 5% of the time spent in the regular phase. Because of that, the total execution time of the coupled simulation does not substantially change between NAIVE and PROJREPART and thus it is not shown here. Remember that we stated before that the coupling time may be up to 30% of the total execution time. This may occur when the number of



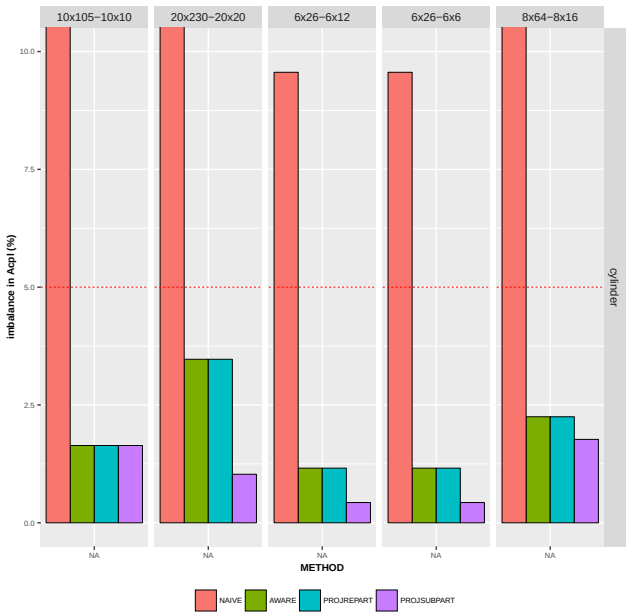
(a) Relative edgecut in G_A.



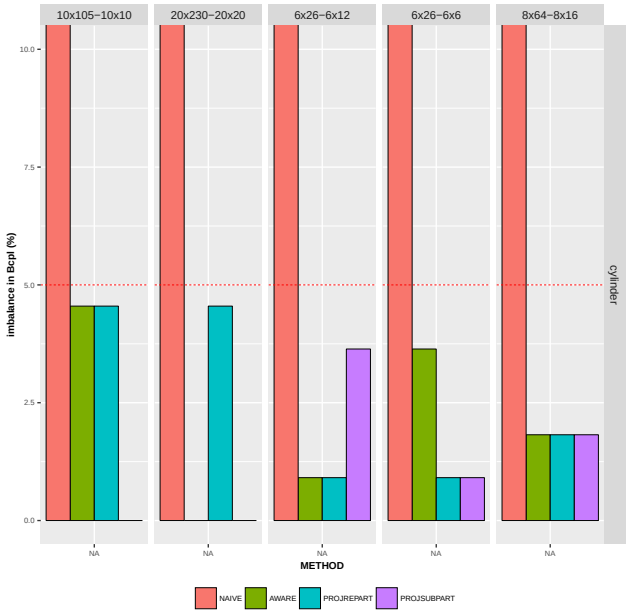
(b) Relative edgecut in G_B.

Figure 4.32: Comparison of co-partitioning algorithms with `RBBGM` method: edgecut results for Naive, Aware and ProjRepart on different values of k_A, k_B. Results for `cylinder`.

k_B become very large. For the `cylinder` component B represents the slower solver (AVBP) and thus when k_B becomes very large, the time of the regular



(a) Imbalance in G'_A.



(b) Imbalance in G'_B.

Figure 4.33: Comparison of co-partitioning algorithms with `RBBGM` method: imbalance results during the coupling phase for Naive, Aware and Proj-Repart on different values of k_A, k_B. Results for `cylinder`.

phase is minimized while the time in the coupling phase remains the same (or even increases). Note that in this study we did not explore very large values

Table 4.5: Results on the execution time of the AA simulation with different input size for the two components (`cylinder` case).

	6×26		8×64		10×105		20×230	
	Naive	ProjRepart	Naive	ProjRepart	Naive	ProjRepart	Naive	ProjRepart
$t_{reg}(ms)$	190	190	370	370	310	310	50	40
$t_{cpl}(ms)$	0.836	0.598	2.102	1.884	2.453	2.175	2.208	2.067
relative gain	-	(29%)	-	(11%)	-	(12%)	-	(7%)

of k_A and k_B.

4.5 Conclusion

In this chapter, we introduce the data distribution problem for coupled simulations, following the classic definition of the graph partitioning problem. To do so, we propose to enrich the classic graph model with fixed vertices, in order to influence the partitioning results between different components or different phases (regular and coupling).

In this context, we present three new co-partitioning algorithms, Aware, ProjRepart and ProjSubpart and we compare them to the currently used approaches Naive and Multiconst. Remember that Naive addresses the load balancing of coupled simulations without considering the coupling between components while Multiconst attempts to simultaneously balance both phases, within each component, using a multi-criteria graph partitioning. Note that neither approach takes into account the inter-component communications.

In this chapter, we evaluate the proposed algorithms in a series of experiments on synthetically generated and real-life data. We divide our experiments in 3 categories, one that determines internal tuning, one that evaluates the scalability as the number of processors increases and one that tests different sizes of coupling interface (from surface to fully volumetric coupling).

We note that our co-partitioning algorithms succeed to balance the computational load in the coupling phase in every case for the surface coupling while Naive and Multiconst fail to do so. However, when the coupling overlap becomes larger, the multiple objectives of the co-partitioning become harder to satisfy. Surprisingly, we see that our algorithms do not highly degrade the global graph edgecut for either component, despite the additional constraints that are imposed to the problem as opposed to the Multiconst approach. We also remark that ProjRepart and ProjSubpart manage to minimize the number of inter-component messages. Regarding the real-life coupled simulation, the obtained results are quit similar. In more details, we notice that the number of processors that are assigned during the coupling phase may highly condition the final partitioning results. Therefore, one may carefully choose the above values in order to not increase the global edgecut,

especially for components where the ratio between the coupling interface and the entire domain is high. Finally, preliminary results lead us to believe that such co-partitioning algorithms may result in a minimized execution time during the inter-component communications of a coupled simulation, but further investigation should be made.

Chapter 5

Conclusion and Future Work

Conclusion

In this work, we study the problem of load balancing for modern coupled simulations that arise in various domains such as computational physics, material science and climate science. Coupled simulations represent complex physical phenomena and consist of a number of different component models, plugged in together under a coupling environment. Typically, within a coupled simulation, different components interact with each other exchanging data on their coupling interfaces. As a result, the computational load of each component changes throughout the execution leading to load imbalance for the coupled simulation. In this context, one should obtain a better data distribution that takes explicitly into account the coupling process and balances the load for the entire coupled application.

In this work, we propose new graph partitioning algorithms that address the above problem, denoted as *co-partitioning* (Chapter 4). The goal here is to efficiently decompose the data of a coupled simulation as part of an interactive system and not just as part of independent components. Our co-partitioning algorithms are designed as a generic approach that could be applied in different coupled problems and potentially be extended in a large number of components. We propose three different algorithms Aware, ProjRepart and ProjSubpart and we compared our results in terms of partitioning quality and runtime performance to the currently used approaches, Naive and Multiconst. Moreover, we performed experiments on synthetically generated data and on a real-life application developed at CERFACS for the optimization of gas turbines. The obtained results on the quality of our co-partitioning methods are encouraging even though further experimental investigation on a larger number of real-life coupled simulations should be included. More precisely, we manage to balance the load during the coupling phase, as opposed to the currently used Naive method that is highly imbalance during coupling and Multiconst that often fails to guarantee balance. As a result the maximum

volume of inter-component communication is equally minimized. An important remark concerning the co-partitioning problem is that there is a trade-off between the minimization of the coupling phase and the internal execution of each component.

Surprisingly, we observe that our proposed algorithms do not highly degrade the global edgecut for either component and thus the internal communication among processors of the same component is still minimized. This is not the case for the MULTICONST method especially as the number of processors increases. Regarding the coupled simulation for the real application, we noticed that one may carefully decide the parameters of the co-partitioning algorithms in order not to increase the global edgecut. More precisely, the number of processors assigned in the coupling interface is an important factor that needs to be determined based on the geometry of the problem and the ratio of the coupling interface compared to the entire domain. Again, we remark that our work on co-partitioning is still theoretical and further investigation should be conducted with different geometries and more coupled simulations that are more or less *coupling-intensive*.

Finally in this work (Chapter 3), we proposed graph partitioning algorithms that support initially fixed vertices. The motivation behind this work is our belief that partitioning solutions for problems with initially fixed vertices may be part of the load balancing for coupled simulations. Under this context, we explained why methods that are based on RB do not provide the best solution for partitioning with initially fixed vertices, and we proposed a new graph partitioning method, named KGGGP, that successfully handles such problems.

In more details, KGGGP is based on a greedy graph growing technique and use the fixed vertices that are present in the graph in order to guide the partitioning procedure. Experimental results on two different configurations of fixed vertices showed that KGGGP handles problems with initially fixed vertices better than the commonly used RB methods. Finally note that KGGGP is proposed as a solution for the variant problem with fixed vertices and RB-based methods are still the most efficient solutions for the classic graph partitioning problem.

Future Work

Since the problem of load balancing for coupled simulations is a rather new research topic, there are many different directions to explore as future work. Our first remark is that the load balancing of coupled simulations is a complicated problem and a more systematic experimental study on different configurations of coupled simulations should be performed. It is interesting to explore the scalability of our co-partitioning algorithms in terms of number of processors but also in terms of number of components. In Chapter 2, we gave some examples of multi-component coupled simulations that use three or more com-

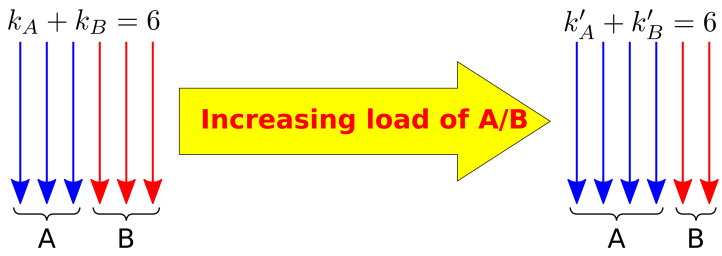


Figure 5.1: Dynamic processor allocation for two components A and B whose relative load A/B increases.

ponents, such as the coupled simulation of earth's climate. Under this context, the data distribution problem of such simulation is more complicated. A question that naturally arises here is whether a many-component coupling can be modeled as a many two-component coupling.

Moreover, a parallel implementation of the co-partitioning algorithms could be an interesting perspective for the future. In this context, an important subproblem is the parallel implementation of KGGGP, where an interesting direction is to apply similar strategies as the ones followed in the parallel BFS schemes [15].

Finally, the resource allocation problem is rather important when a coupled simulation consists of many components. This is critical for the global coupling efficiency, because each component involved in the coupling can be more or less computationally intensive. Consequently, there is an effective trade-off to find between resources assigned to each code in order to avoid that one of them wait for the others at each coupling stage. Another question is what happens if one component becomes more computationally intensive at runtime.

For instance, let us consider a coupled simulation with two components A and B initially running on processors k_A and k_B respectively such that $K = k_A + k_B$. Assuming that one component (say A) becomes highly imbalanced at runtime, it could be convenient to dynamically adapt the number of processors used for each component. Therefore one may perform a repartitioning of the coupled simulation as illustrated on figure 5.1.

Bibliography

[1] Zoltan: Parallel partitioning, load balancing and data-management services. http://www.cs.sandia.gov/Zoltan/Zoltan.html. [Cited on pages vii, 1, and 43]

[2] A. Thevenin A. Piacentini, T. Morel and F. Duchaine. Open-palm: an open source dynamic parallel coupler. In *In IV International Conference on Computational Methods for Coupled Problems in Science and Engineering*, 2011. [Cited on pages vii, 1, and 20]

[3] J. Amaya, E. Collado, B. Cuenot, and T. Poinsot. Coupling LES, radiation and structure in gas turbine simulations. In *Proceedings of the Summer Program*, Center for Turbulence Research, NASA AMES, Stanford University, USA, 2010. [Cited on page 25]

[4] G Anciaux, O Coulaud, and J Roman. High performance multiscale simulation or crack propagation. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 8 pp. –480, 2006. [Cited on page 19]

[5] Cevdet Aykanat, B. Barla Cambazoglu, Ferit Findik, and Tahsin Kurc. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, 67:77–99, January 2007. [Cited on pages 42, 43, and 77]

[6] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 68:609–625, May 2008. [Cited on pages 39, 42, and 43]

[7] DavidA. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer New York, 2014. [Cited on pages ix, 3, 9, and 52]

[8] Roberto Battiti and Alan Bertossi. Differential greedy for the 0-1 equicut problem. In *in Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilities Location*, pages 3–21. American Mathematical Society, 1997. [Cited on pages 10, 46, and 50]

[9] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36(5):570–580, 1987. [Cited on page 6]

[10] S. Berger, S. Richard, F. Duchaine, G. Staffelbach, and L.Y.M. Gicquel. On the sensitivity of a helicopter combustor wall temperature to convective and radiative thermal loads. *Applied Thermal Engineering*, 103(Complete):1450–1459, 2016. [Cited on page 100]

[11] J. T. Betts and W. P. Huffman. Mesh refinement in direct transcription methods for optimal control. *Optimal Control Applications & Methods*, 19:1–21, 1998. [Cited on pages ix, 3, and 5]

[12] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Cambridge University Press, 2009. [Cited on page 5]

[13] Byron A Boville and Peter R Gent. The ncar climate system model, version one*. *Journal of Climate*, 11(6):1115–1130, 1998. [Cited on pages vii, 1, and 20]

[14] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *PPSC*, pages 445–452, 1993. [Cited on pages 13 and 14]

[15] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM. [Cited on page 113]

[16] Andrew E. Caldwell, Andrew B. Kahng, Andrew A. Kennings, and Igor L. Markov. Hypergraph partitioning for VLSI CAD: Methodology for heuristic development, experimentation and reporting. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 349–354, 1999. [Cited on pages ix, 3, and 38]

[17] Paul M. Campbell, Karen D. Devine, Joseph E. Flaherty, Luis G. Gervasio, and James D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003. [Cited on page 6]

[18] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and Lee Ann Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–11, March 2007. [Cited on page 38]

[19] Umit Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, July 1999. [Cited on page 9]

[20] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69(8):711–724, 2009. [Cited on pages 58 and 77]

[21] Jr. Ciarlet, P. and F. Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, 1996. [Cited on page 46]

[22] Anthony P Craig, Mariana Vertenstein, and Robert Jacob. A new flexible coupler for earth system modeling developed for ccsm4 and cesm1. *International Journal of High Performance Computing Applications*, 26(1):31–42, 2012. [Cited on pages 20 and 28]

[23] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989. [Cited on page 6]

[24] John M. Dennis, Mariana Vertenstein, Patrick H. Worley, Arthur A. Mirin, Anthony P. Craig, Robert L. Jacob, and Sheri A. Mickelson. Computational performance of ultra-high-resolution capability in the community earth system model. *IJHPCA*, 26(1):5–16, 2012. [Cited on page 23]

[25] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000. [Cited on pages 46 and 57]

[26] Florent Duchaine, S Jaure, D Poitou, E Quemerais, Gabriel Staffelbach, T Morel, and L Gicquel. High performance conjugate heat transfer with the openpalm coupler. In *V International Conference on Coupled Problems in Science and Engineering*, 2013. [Cited on pages viii and 2]

[27] Alfred E. Dunlop and Brian W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 4(1):92–98, 1985. [Cited on page 38]

[28] Charbel Farhat. A simple and efficient automatic fem domain decomposer. *Computers & Structures*, 28(5):579 – 602, 1988. [Cited on pages 6 and 10]

[29] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *19th Design Automation Conference*, pages 175–181, 1982. [Cited on pages 12 and 46]

[30] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. [Cited on pages vii, 1, and 10]

[31] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. In *Computer Methods in Applied Mechanics and Engineering*, volume 184, pages 485–500, 2000. [Cited on page 38]

[32] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, 2000. [Cited on page 9]

[33] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2), 1995. [Cited on pages 7 and 10]

[34] B. Cuenot J. Amaya, E. Collado and T. Poinsot. Coupling les, radiation and structure in gas turbine simulation. In *Proceedings of the Summer Program Center for Turbulence Research, NASA AMES - Stanford University*, 2019. [Cited on pages 97, 101, and 102]

[35] Sachin Jain, Chaitanya Swamy, and K. Balaji. Greedy algorithms for k-way graph partitioning. In *the 6th international conference on advanced computing*, 1998. [Cited on page 10]

[36] S. Jauré, F. Duchaine, and L. Gicquel. Comparisons of coupling strategies for massively parallel conjugate heat transfer with large eddy simulation. In *In IV International Conference on Computational Methods for Coupled Problems in Science and Engineering*, Kos Island, Greece, 2011. [Cited on pages 71, 97, and 102]

[37] Wolfgang Joppich and M. Kürschner. Mpcci - a tool for the simulation of coupled applications. *Concurrency and Computation: Practice and Experience*, 18(2):183–192, 2006. [Cited on page 20]

[38] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society. [Cited on pages 73 and 85]

[39] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998. [Cited on pages 43 and 51]

[40] George Karypis. METIS, HMETIS, PARMETIS. http://glaros.dtc.umn.edu/gkhome/metis. [Cited on pages vii, 1, 18, and 43]

[41] George Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 56–, New York, NY, USA, 2003. ACM. [Cited on pages 73 and 74]

[42] 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. [Cited on pages 13 and 14]

[43] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, February 1970. [Cited on page 12]

[44] Schloegel Kirk, Karypis George, and Kumar Vipin. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, May 2001. [Cited on page 77]

[45] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. [Cited on page 46]

[46] Jay Larson, Robert Jacob, and Everest Ong. The model coupling toolkit: A new fortran90 toolkit for building multiphysics parallel coupled models. *Int. J. High Perform. Comput. Appl.*, 19(3):277–292, August 2005. [Cited on page 20]

[47] Jay Walter Larson. Ten organising principles for coupling in multiphysics and multiscale models. *ANZIAM Journal*, 48:C1090–C1111, 2009. [Cited on page 68]

[48] R. Leland and B. Hendrickson. A multilevel algorithm for partitioning graphs. In *1995 ACM/IEEE conference on Supercomputing*, 1995. [Cited on pages 6, 10, 13, 14, and 43]

[49] Marius Leordeanu and Martial Hebert. A spectral technique for correspondence problems using pairwise constraints. In *Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*, ICCV '05, pages 1482–1489, Washington, DC, USA, 2005. IEEE Computer Society. [Cited on page 10]

[50] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Accelerating shape optimizing load balancing for parallel fem simulations by algebraic multigrid. In *Proceedings of the 20th International Conference*

on Parallel and Distributed Processing, IPDPS'06, pages 57–57, Washington, DC, USA, 2006. IEEE Computer Society. [Cited on page 11]

[51] Tommy Minyard and Yannis Kallinderis. Parallel load balancing for dynamic execution environments. *Computer Methods in Applied Mechanics and Engineering*, 189:1295–1309, 2000. [Cited on page 6]

[52] Leonid Oliker and Rupak Biswas. Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52:150–177, August 1998. [Cited on page 77]

[53] François Pellegrini. SCOTCH. http://www.labri.fr/perso/pelegrin/scotch/. [Cited on pages vii, 1, 18, and 43]

[54] J.R. Pilkington and S.B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *Parallel and Distributed Systems, IEEE Transactions on*, 7(3):288 –300, March 1996. [Cited on page 6]

[55] Steve Plimpton, Bruce Hendrickson, Steve Attaway, Jeff Swegle, Courtenay Vaughan, and Dave Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society. [Cited on page 73]

[56] Simon Portegies Zwart, Steve McMillan, Breanndán Ó Nualláin, Douglas Heggie, James Lombardi, Piet Hut, Sambaran Banerjee, Houria Belkus, Tassos Fragos, John Fregeau, Michiko Fuji, Evghenii Gaburov, Evert Glebbeek, Derek Groen, Stefan Harfst, Rob Izzard, Mario Jurić, Stephen Justham, Peter Teuben, Joris van Bever, Ofer Yaron, and Marcel Zemp. *A Multiphysics and Multiscale Software Environment for Modeling Astrophysical Systems*, pages 207–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. [Cited on page 25]

[57] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, May 1990. [Cited on page 6]

[58] L. A. Sanchis. Multiple-way network partitioning. 38:62–81, 1989. [Cited on page 17]

[59] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013. [Cited on page 43]

[60] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109 – 124, 1997. [Cited on page 77]

[61] Gualdi S. Bellucci A. Sanna A. Fogli P. G. Manzini E. Vichi M. Oddo P. Scoccimarro, E. and A. Navarra. Effects of tropical cyclones on ocean heat transport in a high resolution coupled general circulation model. *Journal of Climate*, 24:4368–4384, 2011. [Cited on page 20]

[62] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Comp. Sys. Engng*, 2:135–148, 1991. [Cited on page 6]

[63] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM J. Sci. Comput*, 18:1436–1445, 1995. [Cited on pages 16 and 40]

[64] James D. Teresco, Karen D. Devine, and Joseph E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 55–88. 2006. [Cited on page 7]

[65] Lionel Tessé, Francis Dupoirieux, and Jean Taine. Monte carlo modeling of radiative transfer in a turbulent sooty flame. *International Journal of Heat and Mass Transfer*, 47(3):555 – 572, 2004. [Cited on page 97]

[66] S. Valcke. The oasis3 coupler: a european climate modelling community software. *Journal of Geosci. Model Dev.*, pages 373–388, 2013. [Cited on page 20]

[67] Rafael Van Driessche and Dirk Roose. Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 392–397. Springer Berlin / Heidelberg, 1995. [Cited on page 10]

[68] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, January 2005. [Cited on page 43]

[69] Clément Vuchener and Aurélien Esnard. Dynamic Load-Balancing with Variable Number of Processors based on Graph Repartitioning. In *Proceedings of High Performance Computing (HiPC 2012)*, Pune, Inde, 2012. 9 pages. [Cited on pages 77 and 90]

[70] Clément Vuchener and Aurélien Esnard. Graph Repartitioning with both Dynamic Load and Dynamic Processor Allocation. In *International Conference on Parallel Computing - ParCo2013*, Advances of Parallel Computing, pages 243–252, München, Allemagne, 2013. [Cited on page 77]

[71] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comput. Syst*, 17:601–623, 2001. [Cited on page 6]

[72] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. *Applied Mathematical Modelling*, 25(2):123 – 140, 2000. Dynamic load balancing of mesh-based applications on parallel. [Cited on page 74]

[73] M. W. Washington, W. J. Weatherly, A. G. Meehl, J. A. Semtner Jr., W. T. Bettge, P. A. Craig, G. W. Strand Jr., J. Arblaster, B. V. Wayland, R. James, and Y. Zhang. Parallel climate model (pcm) control and transient simulations. *Climate Dynamics*, 16(10):755–774, 2000. [Cited on page 20]

[74] Y Wu, DC Haworth, MF Modest, and B Cuenot. Direct numerical simulation of turbulence/radiation interaction in premixed combustion systems. *Proceedings of the Combustion Institute*, 30(1):639–646, 2005. [Cited on page 97]

[75] Yufang Zhang, Ronan Vicquelin, Olivier Gicquel, and J Taine. Physical study of radiation effects on the boundary layer structure in a turbulent channel flow. *International Journal of Heat and Mass Transfer*, 61:654–666, 2013. [Cited on page 97]

[76] Simon Zimny, Bastien Chopard, Orestis Malaspinas, Eric Lorenz, Kartik Jain, Sabine Roller, and Jörg Bernsdorf. A multiscale approach for the coupled simulation of blood flow and thrombus formation in intracranial aneurysms. *Procedia Computer Science*, 18:1006 – 1015, 2013. [Cited on pages 23 and 24]

[77] Ümit V. Çatalyürek and C. Aykanat. PaToH: A Multilevel Hypergraph Partitioning Tool. http://bmi.osu.edu/~umit/software.html/#patoh, 1999. [Cited on page 43]

Publications

[78] Maria Predari et Aurélien Esnard : Coupling-Aware Graph Partitioning Algorithms: Preliminary Study. *In IEEE International Conference on High Performance Computing (HiPC 2014)*, Goa, India, décembre 2014. [Not cited.]

[79] Maria Predari et Aurélien Esnard : Graph Operators for Coupling-aware Graph Partitioning Algorithms. *In CIMI Workshop on Innovative clustering methods for large graphs and block methods*, Toulouse, France, juillet 2015. [Not cited.]

[80] Maria Predari et Aurélien Esnard : New graph partitioning techniques for load balancing of coupled simulation. womENcourage 2015, septembre 2015. Poster. [Not cited.]

[81] Maria Predari et Aurélien Esnard : A k-way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications. *In 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Parallel, Distributed, and Network-Based Processing (PDP 2016), page 8, heraklion, Greece, février 2016. [Not cited.]

[82] Maria Predari et Aurélien Esnard : Graph partitioning techniques for load balancing of coupled simulations. SIAM Workshop on Combinatorial Scientific Computing (CSC16) , octobre 2016. Poster. [Not cited.]

[83] Maria Predari, Aurélien Esnard et Jean Roman : Comparison of Methods for Graph Partitioning with Initial Fixed Vertices. *In Parallel Computing Journal (under submission)*, 2016. [Not cited.]