Comparison of initial partitioning methods for multilevel direct k-way graph partitioning with fixed vertices

Maria Predari, Aurélien Esnard, Jean Roman

Comparison of Initial Partitioning Methods for Multilevel Direct k-way Graph Partitioning with Fixed Vertices ☆

Maria Predari[a,*], Aurélien Esnard[a], Jean Roman[b]

[a] *Univ. of Bordeaux, Inria, CNRS (LaBRI UMR 5800) F33400 Talence, France*
[b] *Inria, Bordeaux INP, CNRS (LaBRI UMR 5800), F33400 Talence, France*

Abstract

In scientific computing, load balancing is a crucial step conditioning the performance of large-scale applications. In this case, an efficient decomposition of the workload to a number of processors is highly necessary. A common approach to solve this problem is to use graph representation and perform a graph partitioning in k-parts using the multilevel framework and the recursive bisection (RB) paradigm. However, in graph instances where fixed vertices are used to model additional constraints, RB often produces partitions of poor quality.

In this paper, we investigate the difficulties of RB to handle fixed vertices and we compare its results with two different alternatives. The first one, called KGGGP is a direct k-way greedy graph growing partitioning that properly handles fixed vertices while the second one, introduced in KPaToH, uses RB and a post-processing technique to correct the obtained partition. Finally, experimental results on graphs that represent real-life numerical simulations show that both alternative methods provide improved partitions compared to RB.

Keywords: high-performance computing ; graph partitioning ; fixed vertex ; multilevel framework ; parallel simulations.

☆. This is an extended version of a previous work presented in PDP'16 [1]

∗. Corresponding author

Email addresses: maria.predari@inria.fr (Maria Predari), aurelien.esnard@labri.fr (Aurélien Esnard), jean.roman@inria.fr (Jean Roman)

1. Introduction

The increasing complexity of modern applications often dictates a decomposition of the computational load in order to ensure high performance. In other words, when parallel simulations are executed on large-scale systems, a distribution of their workload to a number of available processors is essential. Finding such a decomposition corresponds to the *load balancing* of a parallel application and results in a substantial minimization of the overall execution time.

In literature, many combinatorial problems such as the one described above can be solved in terms of graph modeling [2, 3, 4]. More precisely a vertex of a graph represents a basic computation while an edge represents a dependency between two computations. Besides, each vertex has a weight proportional to the computation's cost, while each edge has a weight representing the communication costs between computations. Therefore, to balance the load of an application among k processors, one may perform a *graph partitioning* in k parts of roughly equal size such that a minimal number of edges connects vertices in different parts. Nowadays, the most common approach to solve the graph partitioning problem is based on the multilevel approach to compress the problem and on the recursive-bisection heuristic to solve it on a smaller instance. Consequently, graph partitioning appears as a fundamental technique for parallel computing.

However, in many applications the classic graph partitioning often fails to capture the true essence of the problem. In such settings, the model can be improved by including the "desire" of each vertex to be present in a certain part at the final solution. This idea has already been expressed in the form of two problems. The first one is the skew graph partitioning that introduces a penalty function in the classic objective formula to represent the notion of

desire [5]. During partitioning the desire function should be satisfied as an additional constraint to the classic ones.

In this paper, we focus on the second formulation which is a variant of the classic graph partitioning problem, the *graph partitioning problem with initially fixed vertices*. This modified instance of graph partitioning typically appears when the underlying application imposes strict 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 part assignment is pre-defined as part of the input description and shall not move throughout the partitioning. On the other hand, vertices that have no particular reason to be in a certain part are referred to as free and may move to any part during partitioning.

In scientific computing, a well-known example where the fixed vertex paradigm occurs is the load balancing of adaptive scientific computations [5]. In such applications, the discretisation of the computational domain changes irregularly at run-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 workload. The additional requirement of repartitioning is to minimize the migration volume of moving data among processors from a former partition to a new one. In this case, the graph is enriched with one fixed vertex per part, along with migration edges connecting each fixed vertex to all free vertices of its respective part. Thus, a good approach to solve the repartitioning problem is to use fixed vertices to model the additional constraint and perform a biased partitioning of the enriched graph, which minimizes migration costs, along with regular communication

costs [6, 7].

Following, we consider the problem of load balancing for multi-physics, multi-scales or multi-phase simulations that emerge in various areas such as computational physics, or hydrology and material science. Multi-physics simulations use coupled models in order to solve complex phenomena. Such an example is a hybrid particle/mesh transient dynamics simulation used to model car crashes and underwater explosions [8]. In such settings, acquiring a load decomposition that simultaneously balance both mesh and particle portions of the computation is challenging. Additionally, multi-phase simulations contain different computational phases separated with global synchronization points. In general, the amount of computations performed by each element of the mesh is different for different phases, leading to complex load balancing.

Note that the traditional graph partitioning model is not effective in such computations, and other strategies have been developed to address this problem. One approach is to distribute the overall computations of a multi-physics simulation by performing two different partitionings (a two-step partitioning) and then combine them together during a communication step [8, 9]. Obviously this approach has a major disadvantage, that is, data must be redistributed between the two partitions at every time-step of the simulation.

A solution to this problem is to introduce fixed vertices in order to use the results of the first phase/partitioning as an influence for the result of the second phase/partitioning. Therefore, fixed vertices may guide the partitioning to better solutions while the data re-distribution among different phases/partitionings may be avoided. Similar solutions have been proposed in [10, 11] in the context of multi-phase and coupled simulations respectively.

Note that another common approach for the load balancing of such simulations is based on the multi-constraint graph partitioning where multiple weights

associated with the computational domain are simultaneously balanced [12, 13]. However we believe that this approach has an important limitation. More precisely, respecting multiple constraints is not straightforward especially when the number of parts increases and often results in highly imbalanced solutions [**?**]. Under this context, the graph partitioning with fixed vertices may be an important venue for the load balancing of large-scale multi-physics simulations.

Finally, graph partitioning with initial fixed vertices may be also used for non-numerical applications, such as the top-down placement technique [14] used in the integrated circuit design. In this case related studies [14, 15] suggest that the presence of fixed vertices represents more accurately additional positioning information, such as the locations of external pin connections, or the probable locations of certain cells in the final placement. As a result fixed vertices can direct the development of parts towards a better solution.

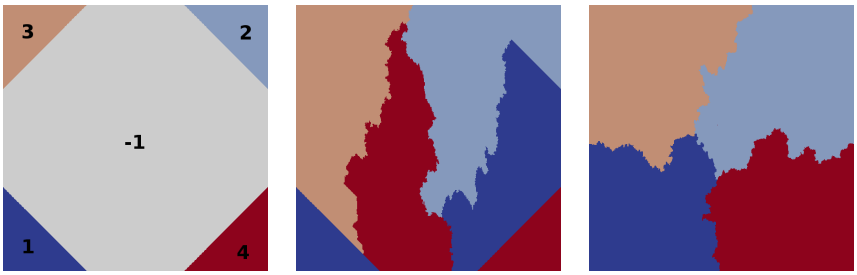
1.1. Issues of Recursive Bisection in Partitioning with Fixed Vertices

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 [16]. Here, we illustrate a simple but compelling example that exhibits the partitioning issues of RB while in Section 3 we attempt to further explain this behavior.

For this example, we use a grid graph with an initial part numbering of fixed vertices such that vertices near the corners are assigned to four different parts following the scheme in Figure 1a. We consider the rest of the vertices as free (part -1) and we partition the graph in four parts. For the partitioning we use two different methods implemented within the same multilevel framework and tuned with the same parameters. To make the comparison of the partitioning phase easier, the refinement algorithms have been disabled for both methods. In Figure 1b, we present the result obtained by the RB method (implemented

by SCOTCH), while in 1c, we present the one obtained by KGGGP.

One may clearly see that under this fixed vertex scheme the partitioning quality of RB is considerably worse than that of KGGGP. We believe that the poor quality of RB is due to the way fixed vertices are pre-assigned to parts. More precisely, there is a conflict of the additional constraints imposed by fixed vertices and the inherent constraints of RB. Note that this example is a particular case of "bad" fixed vertex scheme but it clearly indicates the limitations of RB. Indeed, RB is not robust when fixed vertices are included in the graph and may lead to highly degraded solutions. On the other hand, a method that is based on k-way direct graph growing techniques (for example the KGGGP method) could be more suitable for such problems.



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

FIGURE 1: 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.

The above example is just an illustration of the problematic behavior of RB methods with initial fixed vertices. Further experiments that confirm our observations follow in Section 6.

1.2. Contributions

The main contribution of this paper is a extensive study of the graph partitioning problem with initially fixed vertices and a systematic comparison of different partitioning algorithms that find such solutions. This problem has not

been thoroughly studied, even though it can be found in many scientific areas, as shown earlier in Section 1. Under this context, we compare the KGGGP method, presented by the authors in [1], with the state-of-the-art RB method and the RBBGM algorithm previously introduced in KPATOH [16]. The comparison focuses on the edgecut quality and runtime efficiency when initially fixed vertices occur in the partitioning. Note that, as KPATOH is not publicly available, we provide a new implementation of the RBBGM method for graph structures. This allows a fair comparison of all methods since the overhead introduced by feeding edges to a hypergraph tool (KPATOH) is avoided. Additionally, using our own implementation for both approaches removes the dissimilarities that arise because of different coding practices. Finally, we enrich the experimental section of our preliminary work in [1] with a larger, highly diverse collection of graphs that show that both alternative methods provide improved partitions compared to RB.

The rest of the paper is organized as follows. In Section 2, we start by giving some background definitions and we present the state-of-the-art partitioning techniques as well as existing work for partitioning with fixed vertices. Then, in Section 3, we explain how RB works with fixed vertices and why it sometimes fails to properly solve such problems. In Section 4, we review in details the RBBGM method and in Section 5 we present the KGGGP algorithm. We confirm our observations in Section 6 presenting experiments performed on graphs that represent real numerical applications. Finally, we conclude our results in Section 7.

2. Background Definitions & Related Work

In this section, we first present useful definitions and formal statements concerning the partitioning problem in general and then we review existing

studies about the graph partitioning problem with fixed vertices.

2.1. Graph Definitions

Consider a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. Each vertex $u \in V$ has a weight $w(u)$ representing its computational load, while each edge $e \in E$ has a weight $n(e)$ representing communication costs between different computations. A partition of a graph G is an ensemble of k vertex subsets, $P = (V_1, V_2, \ldots, V_k)$, where : i) each part $V_i, 1 \leq i \leq k$ is a non empty subset of V, ii) parts are pairwise disjoint ($V_i \cap V_j = \emptyset$ for all $1 \leq i, j \leq k$) and iii) the union of all k parts is equal to V. Given a vertex v mapped to part V_i, we note $P[v] = i$ as its part number. The initial part numbering of free vertices is -1 while that of fixed vertices corresponds to their predefined part assignment.

Hence, the graph partitioning problem with edge separator 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 minimizes the total edgecut : $\min \sum_{e \in \mathcal{F}}(n(e))$, where $\mathcal{F} = \{(a, b) \in E, a \in V_i \wedge b \in V_j \wedge i \neq j\}$ subject to the balance constraint $W_i \leq W_{avg}(1 + \epsilon)$ for $i = 1, \ldots, k$. In the above formula, W_i is the sum of the weights of all vertices in part V_i and $W_{avg} = \sum_{u \in V} n(u)/k$ 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%. The total edgecut is a classic metric, known to approximate the total communication volume of a parallel application [17] and is used here as a representative of the partitioning quality.

2.2. Multilevel Framework

Despite the computational complexity of the graph partitioning problem (NP-complete [18]), many heuristic algorithms have been proposed in the past

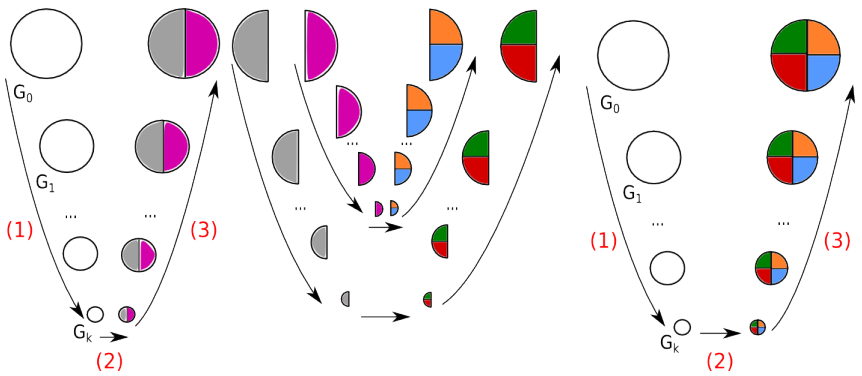
that find reasonably good partitions. Such examples are combinatorial methods that explore local or global solutions working on the graph structure or spectral methods [19] that use algebraic properties to perform the partitioning.

However, nowadays the most common approaches to solve the graph partitioning problem are based on a multilevel framework, where the initial graph is approximated by a sequence of smaller graphs [20]. The main idea of the multilevel framework is to reduce the size of the graph, find a partition for the coarsest graph and project it back to the original one. The most prominent advantage of the multilevel framework is that the size of the graph can be reduced without losing its topological properties and thus the partitioning problem can be solved much faster. The algorithm is divided in three phases : the coarsening, the initial partitioning and the uncoarsening phase. Here, we give a brief description of the multilevel framework, also called *V-cycle*.

During the coarsening phase, a sequence of smaller graphs $G_1, G_2, \ldots, G_t$ is constructed at each level, starting from the original graph $G_0 = (V_0, E_0)$.The goal is to contract edges, merging the adjacent vertices into new super-vertices and update the weights in the coarser graph. The time complexity of a coarsening step is $O(|E|)$ for the mainly used *heavy edge matching* heuristic [21].

During the initial partitioning phase, the coarsest graph is partitioned in the desired number of parts. The algorithm that is used here can be any partitioning algorithm, including bisection methods, spectral methods, greedy methods or geometric ones. The time complexity of this phase is often considered negligible compared to the other phases, since the size of the coarsest graph is smaller ($O(k)$). Nevertheless the result of the initial partitioning phase can be crucial for the final quality of the projected solution back to G_0.

Finally, during the uncoarsening phase each vertex of the finer graph is assigned to the same part as its counterpart vertex in the coarser graph. However



(a) MLRB (W cycle) (b) MLKW (V-cycle)

FIGURE 2: Illustration of a 4-way partitioning with MLRB and MLKW algorithms : (1) coarsening, (2) initial partitioning and (3) uncoarsening phases of the multilevel frameworks.

since finer graphs in the sequence have a higher degree of freedom, allowing further edge cut minimization, we usually apply local refinement algorithms. A class of local refinements that are widely used in partitioning tools is based on the bipartitioning algorithm of Kernighan-Lin (KL) [22] and Fidducia-Mattheyses (FM) [23], known to have a time complexity of $O(|E|)$ for the best implementations.

One of the first methods that used the multilevel scheme is the MLRB algorithm [20] which combines the above idea with the classic recursive bisection. MLRB follows the W cycle paradigm seen in figure 2a where $k-1$ calls to the multilevel framework are performed. Starting from the original graph a call to the multilevel framework is made that divides the graph into two subgraphs. Consecutively, for each resulting subgraph a new multilevel framework is performed and this procedure continues recursively. Note that is required (assuming k is a power of 2) and the time complexity of MLRB is $O(\log(k)|E|)$.

More recently, another class of algorithms, called *multilevel k-way* (MLKW), proposes the use of the multilevel paradigm in order to directly construct a k-way partitioning of a graph, following the V-cycle paradigm shown in figure 2b.

In a V-cycle, the multilevel framework is called just once. and the coarsest graph is now directly partitioned into k parts. Note that refining a k-way partitioning is considerably more complicated than refining a bisection, so the uncoarsening phase of a MLKW algorithm may be more time consuming [21]. Indeed, the time complexity of MLKW is dominated by the complexity of the uncoarsening step which is $O(k|E|)$ in the general case. However clever algorithms that achieve a complexity of $O(|E|)$ exist, such as the one proposed in [21].

2.3. Tools for Partitioning with Fixed Vertices

In Table 1, we present some useful information about common graph and hypergraph partitioning tools, such as SCOTCH, METIS or PATOH. As one may see, more than half of the given tools do not handle at all problems with initially fixed vertices despite the research interest on this problem. Note that this is especially true for graph partitioning tools. Moreover, among the partitioning tools that solve this problem, the majority uses the RB method combined with the multilevel framework (MLRB or MLKW with RB). As we mentioned in Section 1, even though RB is very efficient for the classic graph partitioning problem, it has limitations when fixed vertices incur in the partitioning. However, some tools propose alternative solutions for the above problem and we review them here in detail. We also review RB.

An algorithm that addresses graph problems which involve initially fixed vertices is the one proposed in RM-METIS [6]. RM-METIS is used to solve the k-way repartitioning problem and thus assumes an existing partition that has become imbalanced. As mentioned before, in such re-balancing problems at least one fixed vertex per part is included in the graph in order to model more efficiently the load re-distribution. 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 a

TABLE 1: Graph and hypergraph partitioning tools.

Tools	Type	Fixed	Parallel	Scheme	Initial Part.	Available
METIS [24]	graph	no	no	MLRB	–	source
KMETIS [21]	graph	no	no	MLKW	RB	source
ParMetis [24]	graph	no	yes	MLKW	RB	source
Scotch [25]	graph	yes	no	MLKW	RB	source
PT-Scotch [25]	graph	no	yes	MLRB	–	source
RM-Metis [6]	graph	only k	no	MLKW	greedy	no
KaHIP [26]	graph	no	yes	MLKW	RB	source
Chaco [20]	graph	no	no	MLRB	spectral	source
KGGGP [1]	graph	yes	no	MLKW	greedy	source
HMetis [24]	hypergraph	yes	no	MLRB	–	binary
KHMetis [24]	hypergraph	no	no	MLKW	RB	binary
PATOH [27]	hypergraph	yes	no	MLRB	–	binary
KPATOH [16]	hypergraph	yes	no	MLKW	RBBGM	no*
ZOLTAN (PHG) [28]	hypergraph	yes	yes	MLRB	–	source
Mondriaan [29]	hypergraph	no	no	MLRB	–	source

result the problem resumes in selecting a good shrinking part. A limitation of RM-METIS is that it is designed only for repartitioning and that it may results in a shrinking part of inferior quality (not smooth frontiers). Finally this algorithm is implemented inside the multilevel framework of METIS, but unfortunately its implementation is not publicly available. For the above reasons we choose not to focus on RM-Metis in this study.

Following, we describe in detail the three methods that are part of our experimental study. More precisely we first discuss the RB methodology particularly for problems with fixed vertices (Section 3), then we review the RBBGM algorithm that was first introduced in KPATOH (Section 4) and finally, we present the KGGGP method (Section 5).

3. Recursive Bisection for Fixed Vertices (RB)

Classic RB based methods follow a simple 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) 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. If this step were to be performed in an exhaustive way, RB would have to try $k!$ combinations before choosing the best solution. In other words, there are combinatorially many part numbering selections for each bisection step. Therefore, to avoid choosing among $k!$ combinations, RB decides to blindly group together parts with consecutive part indices, following an inherent rule. 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.

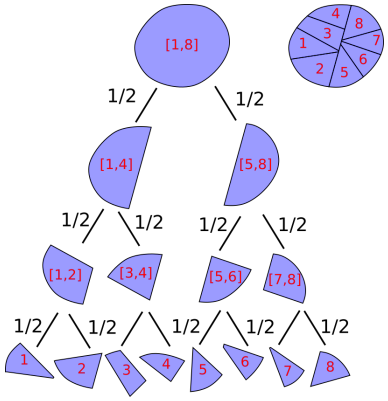


FIGURE 3: Illustration of the RB method : bisection tree and its implicit part numbering (in red) for a 8-way partition.

Moreover, RB can be extended in a straightforward way to include problems with initially fixed vertices. Again, during the first bisection, fixed vertices that are originally pre-assigned to part indices $[1, k/2]$ are fixed to part 1 while fixed vertices that are assigned to parts $[k/2+1, k]$ are considered as fixed to part 2. This idea is applied recursively in every bisection.

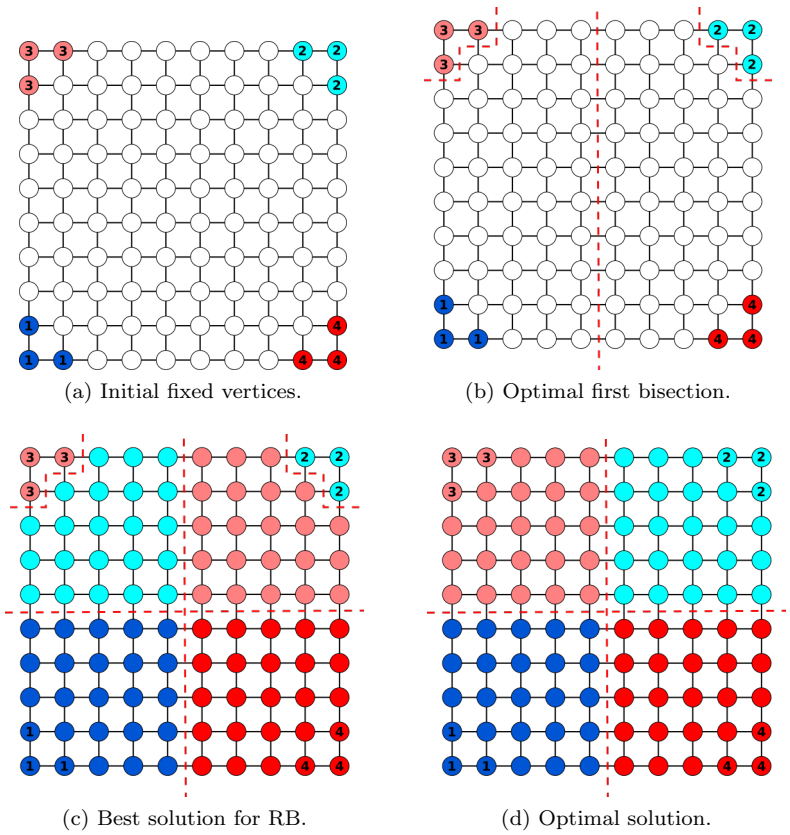
Now let us reconsider the example presented in Section 1 (cf 1.1) where a graph with a "bad" fixed vertex scheme is partitioned in four parts. If we

examine the result of RB, we clearly see that the algorithm cannot select a good bisection between parts $[1, 2]$ and $[3, 4]$ that simultaneously respects the constraint of initial fixed vertices during the first bisection. As a result, the final partitioning quality is considerably poor for the RB based method.

To better understand what goes wrong in this case, let us consider the Figure 4 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 4a. Following a similar methodology as *Simon and Teng* [30], 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 4b (i.e. vertices fixed to part three are disconnected from the part containing vertices fixed to part four and the same happens for vertices fixed to parts two and one). Note that RB cannot find a better bisection that puts fixed vertices of parts one and two 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 4c compared to the optimal solution shown in Figure 4d. 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 6.

4. Recursive Bisection with Bipartite Graph Matching (RBBGM)

An algorithm for hypergraph partitioning is introduced in [16] 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



(a) Initial fixed vertices. (b) Optimal first bisection.

(c) Best solution for RB. (d) Optimal solution.

FIGURE 4: 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.

process, mentioning its inability to explore the combinatorially many part labelings that correspond to a given fixed vertex configuration. To correct the above 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 Mat-

ching [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 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.

1. HCM is the equivalent algorithm of HEM for hypergraph partitioning

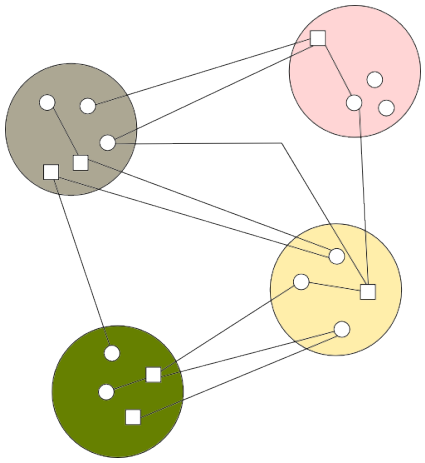
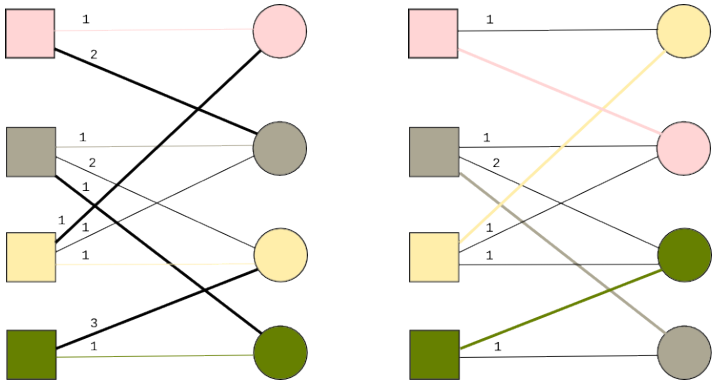


FIGURE 5: Example of coarsest graph where each color corresponds to a different part. Fixed vertices are represented as squares and free ones as circles.

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 5 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 6, we present two instances of the bipartite graph that models the re-assignment of fixed vertices for this example, one (6a) that corresponds to no relabeling and a second one (6b) 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 6b. 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 6: The bipartite graph used for the reassignment of fixed vertices in the example shown in Figure 5. A square vertex represents a set of fixed vertices X_i, while a circle vertex represents a part i

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. Finally note that RBBGM has an important limitation as it does not handle the imbalance issues that may incur if fixed vertices are not evenly distributed to parts.

Below we discuss our implementation of RBBGM for graphs.

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 should 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 [31] which finds an exact solution and has a complexity of $O(k^3)$.

5. 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 successfully handles any number of initially fixed vertices. The algorithm has been initially proposed in [1] where a detailed description is provided. However in order to keep this work self-contained we dedicate here a substantial discuss on KGGGP.

The KGGGP algorithm is an extension of the standard greedy bipartitioning [32, 33] where vertices are iteratively added into two growing parts according to a minimization criterion. KGGGP uses the same idea for a direct k-way

partitioning where k parts (instead of just two) are growing simultaneously, in a concurrent procedure. In a certain way, KGGGP can be seen as a variation of the FM algorithm [23] 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.

5.1. Algorithmic description

Let us consider a graph $G = (V, E)$ and an initial partition P, such that a fixed vertex f to part p is noted as $P[f] = p$ while a free vertex u is noted as $P[u] = -1$. To facilitate our description, we introduce the notion of *displacement* (v, p) where a free vertex v is candidate for moving to a target part p. Each displacement is associated with a gain value, calculated according to a given formula (see Section 5.4) and representing the improvement in terms of edgecut minimization.

In Algorithm 1, we present a brief overview of KGGGP. As one may see, the main steps consist of initializing displacements, selecting displacements and updating the gain for other displacements after a vertex is moved to a part. More precisely, the algorithm starts by computing the gain for all displacements, among all free vertices and all possible parts. Then, at each step of the main loop, the KGGGP algorithm selects the best *global* displacement (v, p), that is, the one having the maximum gain, subject first to the balance constraint ($\mathcal{B}$) and secondly to the connectivity constraint ($\mathcal{C}$). While the first constraint enforces the development of balanced parts, the second constraint ensures that each part remains connected into a single component. In other words, one prefers to select a vertex v in the neighborhood of the growing part p, as far as possible. Once a valid displacement (v, p) is chosen, v moves to the target part p (i.e, $P[v] \leftarrow p$) and all displacements $(v, \star)$ to other parts are removed from consideration. At this point, we need to update the gain of all displacements $(v', \star)$, that involves

vertices v' in the neighborhood of v. The algorithm terminates when there are no more free vertices in P.

Besides, KGGGP uses a similar data structure as in FM adapted here for k parts. This structure is called *gain bucket* and allows a quick locate of the best global displacement. In practice, we use three instances of the gain bucket structure to store and select displacements efficiently : one regular bucket and two additional buckets to manage the displacements that do not respect the connectivity or the balance constraint. In Figure 7, 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.

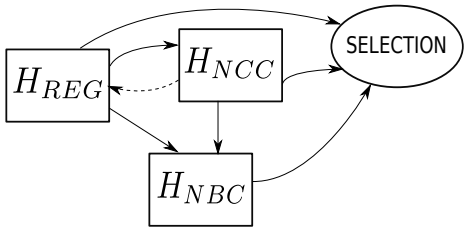


FIGURE 7: Diagram of all possible moves of a displacement during the selection phase. H_{REG} corresponds to the regular gain bucket, while H_{REG} and H_{REG} correspond to the connectivity and balance gain buckets respectively.

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

FIGURE 8: 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 8, 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.

Algorithm 1 The KGGGP algorithm.

```
Inputs : graph G = (V, E)
Inputs/Outputs : partition array P (initialized with fixed and free vertices)
Notations : B = balance constraint ; C = connectivity constraint
compute gain for all possible displacements (v, p) of any free vertex v to any part p
while there are free vertices in P do
    repeat
        consider the displacement d = (v, p) with maximum gain
        if d respects both B and C then
            choose d
        else if d respects B and C cannot be respected anymore then
            choose d
        else if both B and C cannot be respected anymore then
            choose d
        end if
    until a displacement d is chosen
    P[v] ← p
    remove displacements (v, ⋆)
    for all vertex v' adjacent to v do
        update gain of displacements (v', ⋆)
    end for
end while
```

5.2. 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.

5.3. 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 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.

5.4. 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 [33] 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 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 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.

5.5. Time and Space Complexities

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 computes the gain for $k \times |V|$ possible displacements in $O(k|E|)$, as the gain calculation depends on the neighborhood of each vertex. Then, KGGGP selects iteratively $|V|$ displacements and updates the buckets for each displacement in $O(k|E|)$, since it visits the neighborhood of each selected vertex for all parts. Therefore, the total time complexity of KGGGP is $O(k|E|)$. As a reminder, the time complexity of RB is $O(log(k)|E|)$. Finally, the memory complexity is mainly due to saving all possible displacements in the gain bucket data structure, which is $O(k|V|)$.

5.6. 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 [21].

As a result, we do not need in the initialization step to compute *a priori* the gain value for all possible displacements. Instead, after a displacement (v, p) is chosen, we dynamically insert in buckets new displacements (v', p) for all remaining free vertices v' in the neighborhood of v. If there are no more displacements available in buckets 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.

6. Experiments

In this section, we present the experimental results on the partitioning quality and time performance of different algorithms. 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 5. Following, we compare the above methods with our RBBGM implementation of KPATOH (explained in Section 4) and the default RB-based method of SCOTCH (explained in Section 3. Both codes of KGGGP and the one of RBBGM are publicly available in the *MetaPart* library at `http://metapart.gforge.inria.fr`.

All algorithms are implemented inside the same multilevel framework (that of SCOTCH v6.0), as part of the initial partitioning phase of a MLKW method. Moreover, they are tuned with exactly the same partitioning parameters : imbalance tolerance 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 different algorithms 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 this study, we perform three different experiments. First, we present a preliminary experiment that evaluates the best internal parameters for KGGGP_G and KGGGP_L and then we perform two experiments that test our methods on graph instances with initially fixed vertices. Note that a previous study including hypergraph partitioning tools (ZOLTAN and PATOH) has been presented in [1] and provides an insight on the general performance of those tools under the fixed vertex paradigm. Here, we extend the above comparison on a larger collection of

graphs but we restrict our experiments only on partitioning methods for graph structures. Finally, experiments were conducted on a machine with two four-core Intel Xeon CPUs, running at 2.6GHz with 24GB memory. All algorithms are implemented in C and are compiled in *GCC* with -O3 optimization flag.

The dataset used in this study represent real-life applications from different scientific domains (numerical simulations, clustering problems and road networking) available in the public *DIMACS'10* collection [34]. One may find in Table 5 the different graph categories that are used here along with some useful information, such as the total number of vertices or edges, for each graph. Besides, each experiment is performed 5 times for every graph using randomly generated seeds each time.

To thoroughly analyse our results, we include here two different types of figure. In Figures 11 and 13, for each graph category, we present normalized results relative to Scotch on the average values over all included graphs (within that category). 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 as the number of parts increases (Figures 9,10,12). Note that the error bars in those latter figures indicate the standard deviation for each method. Finally, whenever a method fails to respect the balance constraint, its results are not taken into account in the average calculations. Obviously by doing so, we may favor methods that fail to compute a valid partition. To address this problem, we carefully examine the success rate of each partitioning tool in addition to the main metrics.

6.1. Tuning of KGGGP without Multilevel Framework

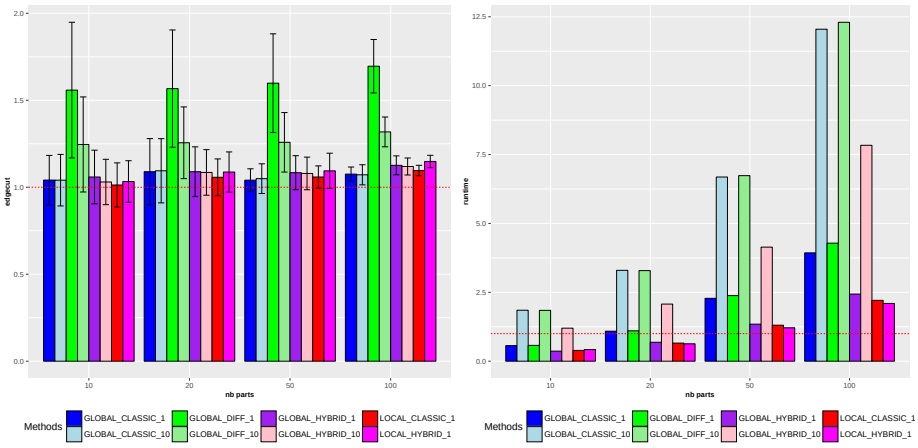
The goal of this preliminary experiment is to tune some important parameters of the KGGGP method. More precisely, we want to evaluate the best gain formula (*classic, diff or hybrid*) and the impact of the *local* optimization (KGGGP_L) vs the *global* approach (KGGGP_G) on both quality and performance. Those parameters have been previously described in Sections 5.4 and 5.6.

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 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.

6.2. Experiments with Fixed Vertices

We present results with fixed vertices from two experimental cases, each one representing a different way to distribute initially fixed vertices to the input graph. 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*. Moreover, in these experiments, we decide to include only the local version of



(a) Edgecut. (b) Runtime.

FIGURE 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.

KGGGP (KGGGP_L) and not the global one. Indeed, based on the results of the previous experiment, KGGGP_L is clearly faster than KGGGP_G while it produces partitions with similar quality.

6.2.1. Bubble Scheme

The idea behind the *bubble* scheme is to create an initial graph structure that already has a number of fixed vertex subsets (bubbles). To make the scheme more generic, we assume that these subsets do not have the same size. We believe that this scheme may represent configurations that come from two-step partitioning methods as explained in Section 1 under the context of multi-physics simulations. To create the bubbles, we compute k seeds that are as distant as possible from each other (based on a BFS [35] technique) and we fix one seed per part randomly. Each seed is later used as the center of a fixed vertex bubble. In particular, each bubble grows using BFS such that neighboring vertices are assigned to the same part as the center of the bubble. The procedure continues

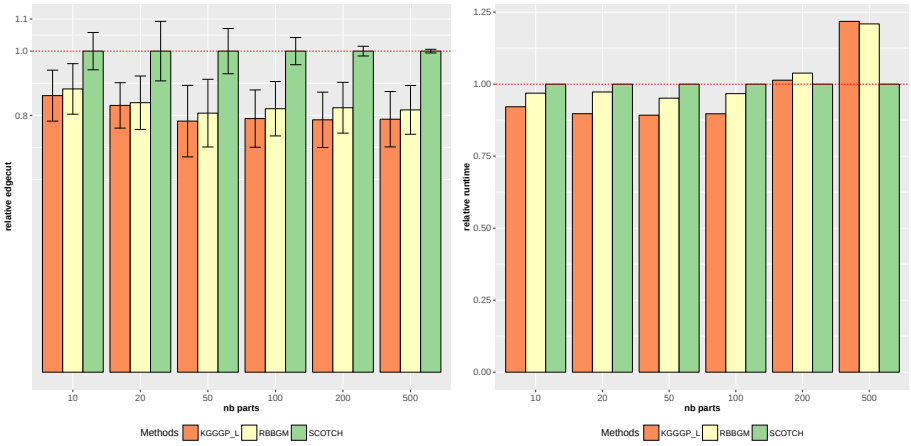
until each bubble reaches a certain percentage of ideal size (assuming vertex weights of 1). In this scheme, we allow different size among the bubbles that vary linearly from 5% to 30% of the ideal part size.

In Figure 10, we depict average results over the entire *DIMACS'10* collection for each method, as the number of parts increases. Figure 10a corresponds to the edgecut while Figure 10b to the partitioning time. In Figure 10a one may see that both KGGGP_L and RBBGM minimize the edgecut compared to SCOTCH with an average gain of 19% and 16% respectively. Concerning the performance results of this experiment, in Figure 10b we see that both KGGGP_L and RBBGM perform, on average, better than SCOTCH until the number of parts reaches 200 (7% faster for KGGGP_L and 2% for RBBGM), but become (21%) slower than SCOTCH when the number of parts is 500.

These results imply a lower scalability for KGGGP_L and RBBGM compared to SCOTCH. For KGGGP_L, this is not a surprise considering its theoretical complexity that depends on the number of parts. Additionally, it is important to note that if a heuristic solution were used instead of the Hungarian algorithm, the time performance of RBBGM might have been optimized. However, in this study we favor an exact solution to the maximum weighted bipartite graph problem, since the performance of this step is not of critical importance.

Moreover in Figure 11, we present the same results but we depict each graph category separately. This representation allow us to evaluate each method upon different graph structures. Additionally, to facilitate the experimental description we group the graph collections in two different categories, namely *group1* and *group2*. More precisely, *group1* includes either graphs that derive from numerical simulations (Matrix, Numerical and Dynframe) or graphs that are deliberately chosen for partitioning (Walshaw). On the other hand *group2* includes graphs from the Clustering and Streets collections that appear to be more dif-

ficult to partition [36]. 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). In Figure 11 we observe that KGGGP_L and RBBGM equally minimize the edgecut for *group1* (with an average gain of 19%) but exhibit different results for *group2*. More precisely we may see that KGGGP_L provides better results compared to RBBGM for *group2* with an average gain of 20% compared to SCOTCH against a 10% gain for RBBGM.



(a) Edgecut. (b) Runtime.

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

Additionally, if we focus on the performance results of *group2*, we see that KGGGP_L is on average 27% faster that SCOTCH and 24% than RBBGM. This leads us to believe that a k-way graph partitioning may be more suitable for such graph structures. On the other hand, if we examine the performance results of *group1* we confirm the scalability issues of KGGGP_L but we also remark that the poor performance of the global results (in Figure 10b) comes mainly from the Walshaw collection when the number of parts is 500. Indeed in this case KGGGP_L is 2.5 times slower than SCOTCH.

Finally, note that both SCOTCH and RBBGM have difficulties finding a parti-

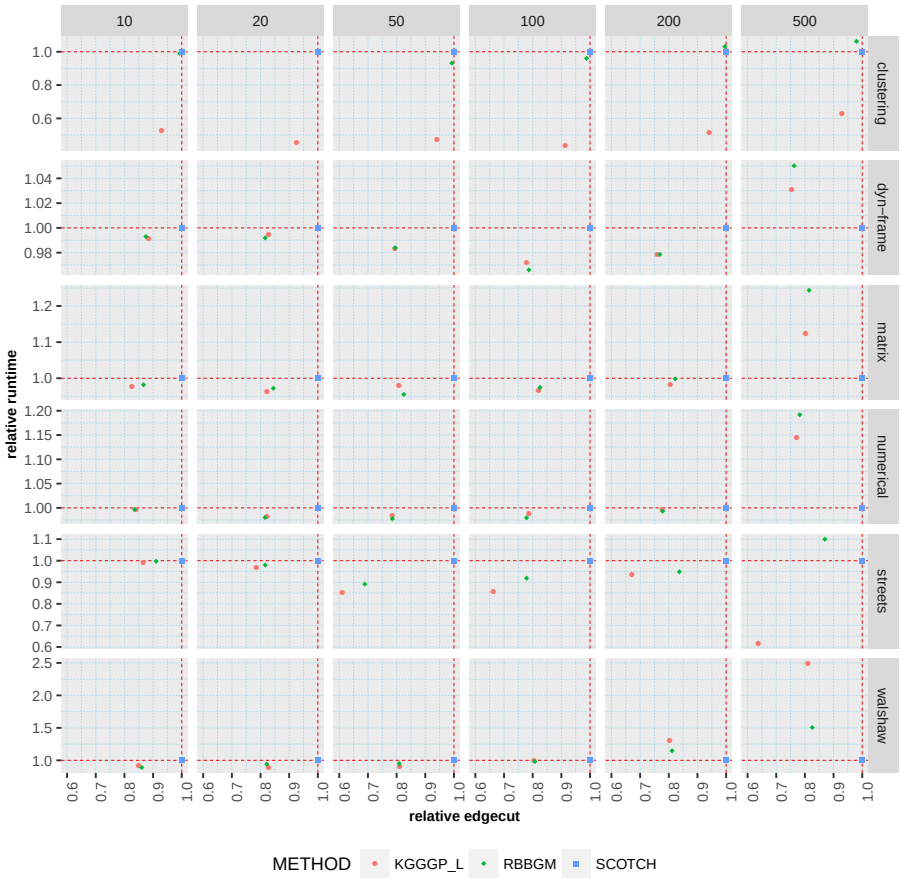


FIGURE 11: Average results on the edgecut quality and time execution for each group in the *DIMACS'10* collection (*bubble* scheme).

tioning solution that respects the imbalance tolerance and often exhibit a large number of failures. In Table 3, we demonstrate the average percentage of failures for each method over all number of parts, depicted for each collection. In this context, we may say that KGGGP_L is more robust since it fails only for the
645 Walshaw collection when the number of parts is 500.

TABLE 3: Average failure percentages for each method per graph collection over all number of parts (*bubble* scheme).

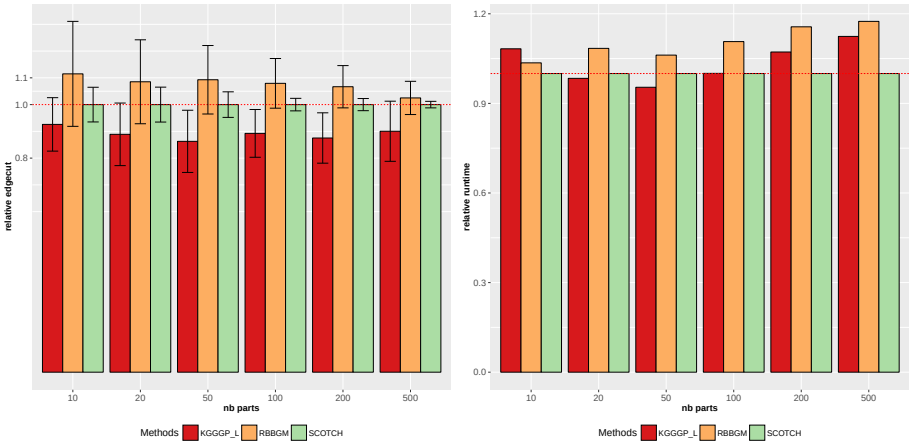
method	streets	matrix	numerical	clustering	dynframe	walshaw
SCOTCH	42%	1%	10%	13%	3%	10%
RBBGM	48%	3%	15%	12%	1%	10%
KGGGP_L	0%	0%	0%	0%	0%	1%

6.2.2. Repart Scheme

For the *repart* scheme, we follow the repartitioning method proposed by ZOLTAN in [7] where an initially balanced partition becomes deliberately imbalanced. Here, the initial partition is obtained with a RB-based method and the size of each part is linearly increased from 0% to 50%, by changing the weight of random vertices in each part. It is important to mention that the part numbering of the initial partition is respected in the imbalanced result. Following, we build an enriched graph of the resulting partition, adding a single fixed vertex per part along with migration edges that connect the fixed vertex with its respective part. In order to enforce the connectivity of fixed vertices with their respective parts, we add a relatively important weight to the migration edges. 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 5 omitting the large collections Dynframe and Streets.

For this experiment, global results on the edgecut and partitioning time are presented in Figure 12 while the same results are also depicted separately for each group of graphs in Figure 13. In Figure 12a, 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. Moreover, as seen in Figure 13, KGGGP_L may reach an edgecut minimization of up to 30% compared to SCOTCH and up to 36% compared to RBBGM for some collections (for instance, the Numerical one). Additionally, we observe that RBBGM does not globally produce partitions of high quality for this experiment and thus may

not be a suitable solution for a scheme such as *repart*. A possible explanation is that the maximum-weight graph matching problem that should be solved here is more complicated due to the large number of migration edges. As a result the additional edgecut of re-introducing fixed vertices to the graph is rather significant. A second explanation is that RBBGM removes the fixed vertices before repartitioning and thus loses the ability to minimizing the data migration between the initially imbalanced input and the final balanced partition.



(a) Edgecut. (b) Runtime.

FIGURE 12: Average results over the entire *DIMACS'10* collection (*repart* scheme).

Regarding the runtime performance of this experiment, in Figure 12b one may see that SCOTCH is on average the fastest method followed by KGGGP_L with an average performance overhead of 2% while RBBGM is the slowest method with 9% overhead compared to SCOTCH. Even though KGGGP is not much slower than SCOTCH, these results confirm again the poor scalability of KGGGP_L as the number of parts increases. However as one may see in Figure 13 for certain group of graphs (Numerical and Matrix), KGGGP_L is faster than SCOTCH even when the number of parts increases. Finally, the average percentage of failures over the number of parts for this experiment can be found in Table 4.

TABLE 4: Average failure percentages for each method per graph collection over all number of parts (*repart* scheme).

method	matrix	numerical	clustering	walshaw
SCOTCH	13%	4%	7%	15%
RBBGM	0%	0%	0%	4%
KGGGP_L	0%	0%	0%	0%

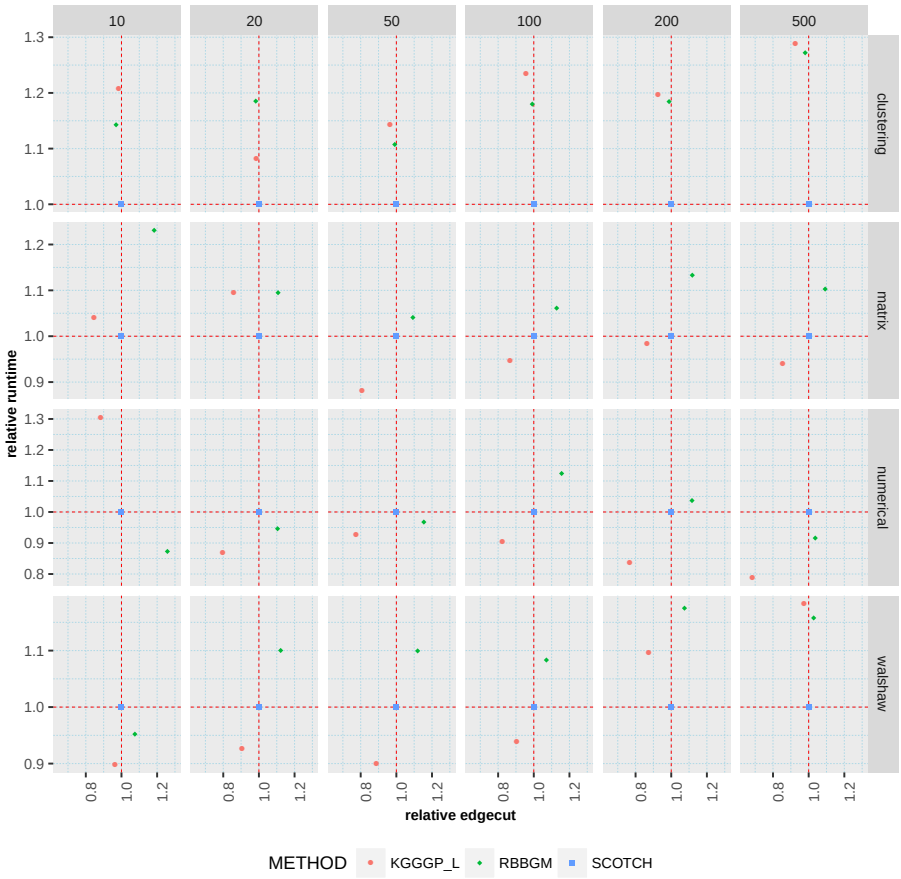


FIGURE 13: Average results on the edgecut quality and time execution for each group in the *DIMACS'10* collection (*repart* scheme).

7. 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. Under this model, additional constraints of the underlying problem are represented in the graph with initially fixed vertices. Unfortunately, in such settings the state-of-the-art algorithm (RB), often does not produce partitions of good quality. Here, we investigate the behavior of RB under this constraint and we present a comparison between RB and the two main alternatives for such problems. More precisely, we compare the KGGGP method that have been previously proposed by the authors and a new implementation of the algorithm used in KPATOH (named RBBGM). This implementation is necessary for a systematic comparison of the algorithms that handle fixed vertices since KPATOH is not publicly available.

In this study, KGGGP and RBBGM are tested on two different configurations of fixed vertices and their results in terms of edgecut minimization and runtime performance indicate that RB is not the optimal solution for such problems. More precisely, in the first experiment KGGGP and RBBGM exhibit a maximum edgecut gain of 40% and 30%, respectively, compared to SCOTCH. In the second experiment KGGGP has a maximum gain of 26% and 35% compared to SCOTCH and RBBGM, respectively. Finally note that KGGGP remains robust to different graph structures which is not the case for SCOTCH or RBBGM, but is susceptible to an increasing number of parts in terms of runtime performance.

Among the perspectives of this work is to apply the KGGGP algorithm for load-balancing of complex multi-physics simulations. In such configurations fixed vertices may influence the partitioning results between different physical components improving the overall partitioning quality of the solution.

Acknowledgment

The authors would like to thank Bora Uçar for his valuable advice for the implementation of the RBBGM method used in this study.

References

[1] M. Predari, A. 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, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016, 2016, pp. 280–287.

[2] J. D. Teresco, K. D. Devine, J. 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, Vol. 51, 2006, pp. 55–88.

[3] N. J. Dingle, P. G. Harrison, W. J. Knottenbelt, Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large markov models, J. Parallel Distrib. Comput. 64 (8) (2004) 908–920.

[4] B. Hendrickson, R. W. Leland, R. V. Driessche, Enhancing data locality by using terminal propagation., in : HICSS (1), 1996, pp. 565–574.

[5] B. Hendrickson, R. W. Leland, R. V. Driessche, Skewed graph partitioning, in : Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[6] C. Aykanat, B. B. Cambazoglu, F. Findik, T. Kurc, Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids, J. Parallel Distrib. Comput. 67 (2007) 77–99.

[7] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, L. A. Riesen, A repartitioning hypergraph model for dynamic load balancing, J. Parallel Distrib. Comput. 69 (8) (2009) 711–724.

[8] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, D. Gardner, Parallel transient dynamics simulations, J. Parallel Distrib. Comput. 50 (1) (1998) 104–122.

[9] K. Brown, S. Attaway, S. Plimpton, B. Hendrickson, Parallel strategies for crash and impact simulations, Computer Methods in Applied Mechanics and Engineering 184 (2–4) (2000) 375–390.

[10] C. Walshaw, M. Cross, K. McManus, Multiphase mesh partitioning, Applied Mathematical Modelling 25 (2) (2000) 123 – 140, dynamic load balancing of mesh-based applications on parallel.

[11] M. Predari, A. Esnard, Coupling-aware graph partitioning algorithms : Preliminary study, in : 21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014, 1–10, 2014.

[12] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint graph partitioning, in : Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 1–13.

[13] G. Karypis, Multi-constraint mesh partitioning for contact/impact computations, in : Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC '03, ACM, New York, NY, USA, 2003, pp. 56–.

[14] P. Maria, Load balancing for parallel coupled simulations, Ph.D. thesis, University of Bordeaux (2016).

[15] A. E. Caldwell, A. B. Kahng, A. A. Kennings, I. 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, 1999, pp. 349–354.

[16] A. E. Dunlop, B. W. Kernighan, A procedure for placement of standard-cell VLSI circuits., IEEE Trans. on CAD of Integrated Circuits and Systems 4 (1) (1985) 92–98.

[17] C. Aykanat, B. B. Cambazoglu, B. Uçar, Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices, J. Parallel Distrib. Comput. 68 (2008) 609–625.

[18] B. Hendrickson, T. G. Kolda, Graph partitioning models for parallel computing, Parallel Comput. 26 (12) (2000) 1519–1534.

[19] M. R. Garey, D. S. Johnson, Computers and Intractibility : A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[20] B. Hendrickson, R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations, SIAM J. Sci. Comput. 16 (2).

[21] R. Leland, B. Hendrickson, A multilevel algorithm for partitioning graphs, in : 1995 ACM/IEEE conference on Supercomputing, 1995.

[22] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed Computing 48 (1998) 96–129.

[23] B. W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell System Technical Journal 49 (1970) 291–307.

[24] C. M. Fiduccia, R. M. Mattheyses, A linear-time heuristic for improving network partitions, 19th Design Automation Conference (1982) 175–181.

[25] G. Karypis, METIS, HMETIS, PARMETIS, `http://glaros.dtc.umn.edu/gkhome/metis`.

[26] F. Pellegrini, SCOTCH, `http://www.labri.fr/perso/pelegrin/scotch/`.

[27] P. Sanders, C. Schulz, Think Locally, Act Globally : Highly Balanced Graph Partitioning, in : Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Vol. 7933 of LNCS, Springer, 2013, pp. 164–175.

[28] U. V. Catalyurek, C. Aykanat, PaToH : A Multilevel Hypergraph Partitioning Tool, `http://bmi.osu.edu/~umit/software.html` (1999).

[29] Zoltan : Parallel partitioning, load balancing and data-management services, `http://www.cs.sandia.gov/Zoltan/Zoltan.html`.

[30] B. Vastenhouw, R. H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Rev. 47 (1) (2005) 67–95.

[31] H. D. Simon, S.-H. Teng, How good is recursive bisection ?, SIAM J. Sci. Comput 18 (1995) 1436–1445.

[32] H. W. Kuhn, The hungarian method for the assignment problem, Naval Research Logistics Quarterly 2 (1-2) (1955) 83–97.

[33] J. Ciarlet, P., F. Lamour, On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint, Numerical Algorithms 12 (1) (1996) 193–214.

[34] R. Battiti, A. Bertossi, Differential greedy for the 0-1 equicut problem, in : in Proceedings of the DIMACS Workshop on Network Design : Connectivity and Facilities Location, American Mathematical Society, 1997, pp. 3–21.

[35] D. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, D. Wagner, Benchmarking for graph clustering and partitioning, in : R. Alhajj, J. Rokne (Eds.), Encyclopedia of Social Network Analysis and Mining, Springer New

York, 2014, pp. 73–82.
URL http://www.cc.gatech.edu/dimacs10

[36] R. Diekmann, R. Preis, F. Schlimbach, C. Walshaw, Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM, Parallel Computing 26 (12) (2000) 1555–1581.

[37] S. Rajamanickam, E. G. Boman, Parallel partitioning with zoltan : Is hypergraph partitioning worth it ?, in : Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, 2012, pp. 37–52.

TABLE 5: List of graphs used for experiments from the popular DIMACS'10 collection.

					degree		
group	**collection**	**graph**	**# vtx**	**# edges**	**avg**	**min**	**max**
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	4.00	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