1 Variable partition inertia: graph repartitioning and load-balancing for adaptive meshes

CHRIS WALSHAW

Computing & Mathematical Sciences, University of Greenwich,
Old Royal Naval College, London, SE10 9LS, UK.

1.1 INTRODUCTION

Graph-partitioning is now well established as an important enabling technology for mapping unstructured meshes, for example from applications such as Computational Fluid Dynamics (CFD), onto parallel machines. Typically, the graph represents the computational load and data dependencies in the mesh, and the aim is to distribute it so that each processor has an equal share of the load whilst ensuring that communication overhead is kept to a minimum.

One particularly important variant of the problem arises from applications in which the computational load varies throughout the evolution of the solution. For example, heterogeneity in either the computing resources (e.g. processors which are not dedicated to single users) or in the solver (e.g. solving for fluid flow and solid stress in different regions during a multiphysics solidification simulation, e.g. [13]) can result in load-imbalance and poor performance. Alternatively, time-dependent mesh codes which use adaptive refinement can give rise to a series of meshes in which the position and density of the data points varies dramatically over the course of a simulation and which may need to be frequently repartitioned for maximum parallel efficiency.

This dynamic partitioning problem has not been nearly as thoroughly studied as the static problem but an interesting overview can be found in [6]. In particular, the problem calls for parallel load-balancing (i.e. *in situ* on the parallel machine, rather than the bottleneck of transferring it back to some host processor) and a number of software packages, most notably JOSTLE [26], and ParMETIS [15], have been developed and can compute high quality parti-

tions, in parallel, using the existing (unbalanced) partition as a starting point. A question arises, however, over data migration – in general, the better the computed partition, the more data (mesh elements, solution variables, etc.) have to be transferred to realize it. Of course either a poor partition or heavy data migration slows the solver down and so the trade-off between partition quality and data migration can be crucial. Furthermore, the requirements of this trade-off may change as the simulation continues (see Section 1.3).

In this chapter, we look at a new framework for managing this trade-off. First, however, we establish some notation.

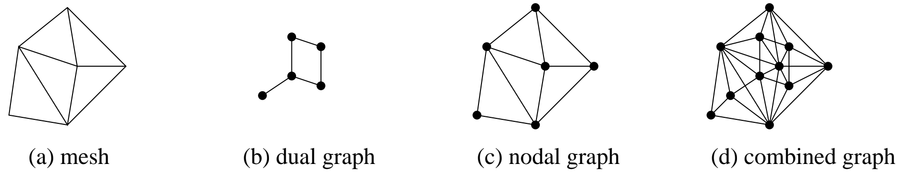
1.1.1 Notation and definitions

Let $G = G(V, E)$ be an undirected graph of vertices, V, with edges, E, which represent the data dependencies in the mesh. We assume that both vertices and edges can be weighted (with non-negative integer values) and that $||v||$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, define a partition π to be a mapping of V into P disjoint subdomains, S_p, such that $\bigcup_P S_p = V$. The weight of a subdomain is just the sum of the weights of the vertices assigned to it, $||S_p|| = \sum_{v \in S_p} ||v||$ and we denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c. Finally, note that we use the words subdomain and processor more or less interchangeably: the mesh is partitioned into P subdomains; each subdomain, S_p, is assigned to a processor p and each processor p owns a subdomain S_p.

The definition of the graph-partitioning problem is to find a partition which evenly balances the load (i.e. vertex weight) in each subdomain, whilst minimising the communication cost. To evenly balance the load, the optimal subdomain weight is given by $\overline{S} := \lceil ||V||/P \rceil$ (where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x) and the **imbalance**, θ, is then defined as the maximum subdomain weight divided by the optimal (since the computational speed of the underlying application is determined by the most heavily weighted processor). Note that $\theta \geq 1$ and perfect balance is given by $\theta = 1$. As is usual, throughout this chapter the communications cost will be estimated by $||E_c||$, the weight of cut edges or cut-weight (although see [23] for further discussion on this point). A more precise definition of the graph-partitioning problem is therefore to find π such that $||S_p|| \leq \overline{S}$ and such that $||E_c||$ is minimized.

The additional objective for dynamic repartitioning is to minimize the amount of data that the underlying application will have to transfer. We model this by attempting to minimize $|V_m|$, the number of vertices which have to migrate (change subdomains) in order to realize the final partition from the initial one.

1.1.2 Mesh-partitioning



(a) mesh (b) dual graph (c) nodal graph (d) combined graph

Fig. 1.1 An example mesh and some possible graph representations.

As mentioned above, many of the applications for which partitioning is used involve a parallel simulation, solved on an unstructured mesh which consists of elements, nodes and faces, etc. For the purposes of partitioning, it is normal to represent the mesh as a graph. Thus, if we consider the mesh shown in Figure 1.1(a), the graph vertices can either represent the mesh elements (the dual graph), Figure 1.1(b), the mesh nodes (the nodal graph), Figure 1.1(c), a combination of both (the full or combined graph), Figure 1.1(d), or even some special purpose representation to model more complicated interactions in the mesh. In each case the graph vertices represent units of workload that exist in the underlying solver and edges represent data dependencies (e.g. the value of the solution variable in a given element will depend on those in its neighboring elements).

1.1.3 Overview

In this chapter we discuss a new framework for the (re)partitioning and load-balancing of adaptive unstructured meshes. In Section 1.2 we first give an overview of the standard graph-partitioning and load-balancing tools and in particular our implementation. In Section 1.3 we then describe the new approach, Variable Partition Inertia, and look at how it relates to previous work. We test the new framework in Section 1.4 and illustrate some of the benefits, and finally in Section 1.5 we present some conclusions and suggest some future work.

Note that for the purposes of this chapter, particularly with regard to the results, we tend to concentrate on situations where the changes in load are at discrete points during the evolution of the solution. This is likely to happen when either the mesh changes (as is the case for adaptive refinement) or the computational resources change. However, the techniques discussed apply equally to situations where the load changes are continuous (or at least quasi-continuous) such as the solidification example mentioned above. In this sort of problem, a further issue is *when* to rebalance, the decision being based on a trade-off between the additional overhead for carrying out the repartitioning and resultant data migration, as against the inefficiency of continuing the

simulation with an unbalanced solver. We do not address that issue here but an algorithm for determining whether or not a rebalance is likely to be profitable (and thus for deciding the frequency of repartitioning) can be found in [1].

1.2 MULTILEVEL REFINEMENT FOR GRAPH-REPARTITIONING

In this section give an overview of the standard graph-(re)partitioning and load-balancing tools and in particular their implementation within JOSTLE, the parallel graph-partitioning software package written at the University of Greenwich [24].

JOSTLE uses a multilevel refinement strategy. Typically such multilevel schemes match and coalesce pairs of adjacent vertices to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (possibly with a crude algorithm) and the partition is successively refined on all the graphs starting with the coarsest and ending with the original. At each change of levels, the final partition of the coarser graph is used to give the initial partition for the next level down. The use of multilevel refinement for partitioning was first proposed by both Hendrickson and Leland [7] and Bui and Jones [4], and was inspired by Barnard and Simon [2], who used a multilevel numerical algorithm to speed up spectral partitioning.

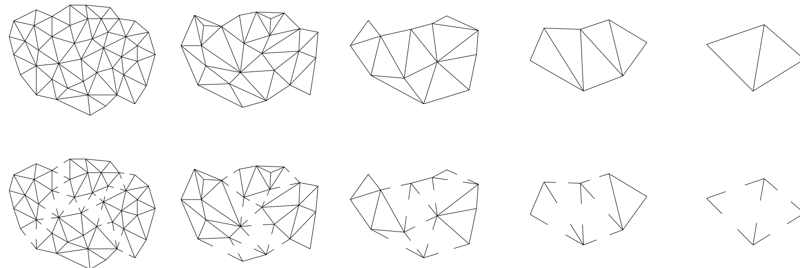


Fig. 1.2 An example of multilevel partitioning

Figure 1.2 shows an example of a multilevel partitioning scheme in action. On the top row (left to right) the graph is coarsened down to 4 vertices which are (trivially) partitioned into 4 sets (bottom right). The solution is then successively extended and refined (right to left). Although at each level the refinement is only local in nature, a high quality partition is still achieved.

The graph-partitioning problem was the first combinatorial optimization problem to which the multilevel paradigm was applied and there is now a considerable body of literature about multilevel partitioning algorithms. Ini-

tially used as an effective way of speeding up partitioning schemes, it was soon recognized as, more importantly, giving them a 'global' perspective [10], and has been successfully developed as a strategy for overcoming the localized nature of the Kernighan-Lin (KL) [12], and other optimization algorithms. In fact, as discussed in [24, §3.2], this coarsening has the effect of *filtering* out most of the poor quality partitions from the solution space, allowing the refinement algorithms to focus on solving smaller, simpler problems.

This very successful strategy, and the powerful abilities of the multilevel framework, have since been extended to other combinatorial problems, such as the travelling salesman problem, e.g. [19].

1.2.1 Multilevel framework

1.2.1.1 Graph coarsening A common method for creating a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ is the edge contraction algorithm proposed by Hendrickson and Leland [7]. The idea is to find a maximal independent subset of graph edges, or a **matching** of vertices, and then collapse them. The set is independent if no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal if no more edges can be added to the set without breaking the independence criterion. Having found such a set, each selected edge, $(v_1, v_2) \in E_l$ say, is collapsed and the vertices, $v_1, v_2 \in V_l$, are merged to form a new vertex, $v \in V_{l+1}$, with weight $\|v\| = \|v_1\| + \|v_2\|$. Edges which have not been collapsed are inherited by the child graph, G_{l+1}, and, where they become duplicated, are merged with their weight combined. This occurs if, for example, the edges (v_1, v_3) and (v_2, v_3) exist when edge (v_1, v_2) is collapsed. Because of the inheritance properties of this algorithm, it is easy to see that the total graph weight remains the same, $\|V_{l+1}\| = \|V_l\|$, and the total edge weight is reduced by an amount equal to the weight of the collapsed edges.

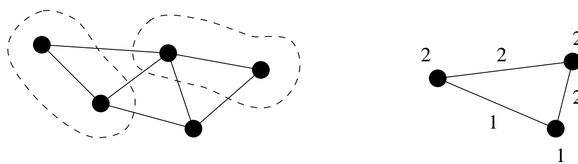


Fig. 1.3 An example of coarsening via matching and contraction

Figure 1.3 shows an example of this; on the left two pairs of vertices are matched (indicated by dotted rings). On the right, the graph arising from the contraction of this matching is shown with numbers illustrating the resulting vertex and edge weights (assuming that the original graph had unit weights).

A simple way to construct a maximal independent subset of edges is to create a randomly ordered list of the vertices and visit them in turn, match-

ing each unmatched vertex with an unmatched neighbor (or with itself if no unmatched neighbors exist). Matched vertices are removed from the list. If there are several unmatched neighbors the choice of which to match with can be random, but it has been shown by Karypis and Kumar [10], that it can be beneficial to the optimization to collapse the most heavily weighted edges.

JOSTLE uses a similar scheme, matching across the heaviest edges, or, in the event of a tie, matching a vertex to the neighbor with the lowest degree (with the aim of trying to avoid highly connected vertices).

In the case of *repartitioning*, an initial partition already exists and it is quite common to restrict the coarsening to use only **local matching** (i.e. vertices are only allowed to match with other vertices in the same subdomain). It has been shown that on its own, this can help reduce vertex migration [26], and when combined with modifications to the refinement algorithms, can be used to help control the trade-off between migration and cut-weight [15]. However, in §1.3.1 and following, we shall see that this is not necessarily the most effective strategy.

Note that even if non-local matching is allowed, a initial partition of each graph level can still be maintained by migrating one of each pair of non-locally matched vertices to the other's subdomain (the choice of which to migrate being based on relative subdomain weights, or even random). JOSTLE allows either local or non-local matching to take place (chosen by the user at run-time).

1.2.1.2 The initial partition The hierarchy of graphs is constructed recursively until the number of vertices in the coarsest graph is smaller than some threshold and then an initial partition is found for the coarsest graph. Since the vertices of the coarsest graph are generally inhomogeneous in weight, some mechanism is then required for ensuring that the partition is balanced, i.e. each subdomain has (approximately) the same vertex weight. Various methods have been proposed for achieving this, often by terminating the contraction so that the coarsest graph, G_L, still retains enough vertices, $|V_L|$, to achieve a balanced initial partition (i.e. so that typically $|V_L| \gg P$) [7, 10]. Alternatively, if load-balancing techniques are incorporated alongside the refinement algorithm, as is the case with JOSTLE, [20], the contraction can be terminated when the number of vertices in the coarsest graph is the same as the number of subdomains required, P, and then vertex v_p is assigned to subdomain S_p, $p = 1, \ldots, P$.

In the case of repartitioning, even this is unnecessary. If only local matching is allowed, then the vertices of the coarsest graph will already be assigned to their original subdomain, and the partition cut-weight of the coarsest graph will match that of the initial partition. It has also been shown [22, 26], that a simple scheme for trading-off vertex migration against cut-weight is to terminate the coarsening early, such as when the number of vertices in the graph falls below some threshold (e.g. in experiments in [22], $20P$ was chosen as an appropriate threshold, where P is the number of processors/subdomains).

The reason for this is simple; each vertex in the coarsest graphs may represent hundreds or even thousands of vertices in the original graph and so moving them from subdomain to subdomain may give rise to very high data migration in the application. Conversely, since coarsening provides a continuum that affords a global perspective, [10, 19], to the refinement algorithms (with single-level refinement having almost no global abilities), then the more coarsening that occurs, the better the cut-weight.

For the methods described here, however, we use our standard technique of coarsening the graph down to P vertices, one per subdomain.

1.2.1.3 Partition extension Having refined the partition on a graph G_{l+1}, the partition must be extended onto its parent G_l. The extension algorithm is trivial; if a vertex $v \in V_{l+1}$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_l$, are also assigned to S_p.

1.2.2 Refinement

At each level, the new partition, extended from the previous level, is refined. Because of the power of the multilevel framework, the refinement scheme can be anything from simple greedy optimization, to a much more sophisticated one, such as the Kernighan-Lin algorithm. Indeed, in principle any iterative refinement scheme can be used and examples of multilevel partitioning implementations exist for simulated annealing, tabu search, genetic algorithms, cooperative search and even ant colony optimization (see [19] for references).

1.2.2.1 Greedy refinement Various refinement schemes have been successfully used including greedy refinement, a steepest descent approach, which is allowed a small imbalance in the partition (typically 3-5%) and transfers border vertices from one subdomain to another if either (a) the move improves the cost without exceeding the allowed imbalance; or (b) the move improves the balance without changing the cost. Although this scheme cannot guarantee perfect balancing, it has been applied to very good effect [11], and is extremely fast.

Although not the default behavior, JOSTLE includes a greedy refinement scheme, accessed by turning off the hill-climbing abilities of the optimization (see below).

1.2.2.2 The k-way Kernighan-Lin algorithm A more sophisticated class of refinement method is based on the Kernighan-Lin (KL) bisection optimization algorithm [12], which includes some limited hill-climbing abilities to enable it to escape from local minima. This has been extended to k-way partitioning (here k is the same as P, the number of processors/subdomains) in different ways by several authors (e.g. [7, 11, 20]) and recent implementations almost universally use the linear time complexity improvements (e.g. bucket sorting of vertices) introduced by Fiduccia and Mattheyses [5].

A typical KL-type algorithm will have inner and outer iterative loops with the outer loop terminating when no vertex transfers take place during an inner loop. It is initialized by calculating the **gain** – the potential improvement in the cost function (the cut-weight) – for all border vertices. The inner loop proceeds by examining candidate vertices, highest gain first, and if the candidate vertex is found to be acceptable (i.e. it does not overly upset the load-balance), it is transferred. Its neighbors have their gains updated and, if not already tested in the current iteration of the outer loop, join the set of candidate vertices.

The KL hill-climbing strategy allows the transfer of vertices between subdomains to be accepted even if it degrades the partition quality and later, based on the subsequent evolution of the partition, the transfers are either rejected or confirmed. During each pass through the inner loop, a record of the best partition achieved by transferring vertices within that loop is maintained, together with a list of vertices which have been transferred since that value was attained. If, during subsequent transfers, a better partition is found, then the transfer is confirmed and the list is reset.

This inner loop terminates when a specified number of candidate vertices have been examined without improvement in the cost function. This number (i.e. the maximum number of continuous failed iterations of the inner loop) can provide a user specified intensity for the search, λ. Note that if $\lambda = 0$ then the refinement is purely greedy in nature (as mentioned in §1.2.2.1). Once the inner loop is terminated, any vertices remaining in the list (vertices whose transfer has not been confirmed) are transferred back to the subdomains they came from when the best cost was achieved.

JOSTLE uses just such a refinement algorithm [20], modified to allow for weighted graphs (even if the original graph is not weighted, coarsened versions will always have weights attached to both vertices and edges). It incorporates a balancing flow of vertex weight, calculated by a diffusive type load-balancing algorithm [9], and indeed, by relaxing the balance constraint on the coarser levels and tightening it up gradually as uncoarsening progresses, the resulting partition quality is often enhanced [20].

Further details of the diffusive load-balancing and how it is incorporated into the refinement can be found in [22]. However, since we regard it as only incidental to the framework described below, and, in particular, since we do not consider the partition inertia scheme to be diffusive load-balancing in its classical sense, we do not discuss it further.

1.2.3 Parallelization

Although not necessarily easy to achieve, all the techniques above have been successfully parallelized. Indeed, the matching, coarsening and expansion components of the multilevel framework are inherently localized and hence straightforward to implement in parallel. The refinement schemes are more difficult, but, for example, by treating each inter-subdomain interface as a sep-

arate problem and then using one of the two processors that shares ownership of the region to run the (serial) refinement scheme above, parallel partitioning has not only been realized, but has also been shown to provide almost identical results (qualitatively speaking) as the serial scheme. More details can be found in [21], and parallel run times for dynamic diffusive repartitioning schemes appear in [26].

1.2.4 Iterated multilevel partitioning

The multilevel procedure usually produces high quality partitions very rapidly, and if extra time is available to the algorithm, then one possibility is to increase the search intensity, λ (see §1.2.2). However this has limited effect and it has been shown, e.g. [19], that an even more effective, although time-consuming technique is to iterate the multilevel process by repeatedly coarsening and uncoarsening and, at each iteration, using the current solution as a starting point to construct the next hierarchy of graphs. When used with local matching, the multilevel refinement will then find a new partition that is no worse than the initial one. However, if the matching includes a random factor, each coarsening phase is very likely to give a different hierarchy of graphs to previous iterations and hence allow the refinement algorithm to visit different solutions in the search space.

We refer to this process, which is analogous to the use of V-cycles in multigrid, as an **iterated multilevel** (IML) algorithm.

1.3 VARIABLE PARTITION INERTIA

A considerable body of work has now arisen on repartitioning schemes for adaptive meshes (e.g. [3, 15, 17, 26, 22]) and tends to resolve into two different approaches. If the mesh has not changed too much then it is generally held that the best way of minimising data migration is to spread the load out diffusively from overloaded to underloaded processors, e.g. [17, 22]. However, if the mesh has changed dramatically then diffusion may not only severely compromise partition quality (since migrating vertices are channelled through certain processors [17]), but may even result in heavy data migration. In such cases it seems more natural to repartition from scratch and then use heuristics to map the new subdomains so as to maximize overlaps with current subdomains as far as possible. This idea, known as **scratch-remapping** has been investigated by Biswas & Oliker [3], who devised appropriate mapping heuristics and improved by Schloegel *et al.* [16], who modified the strategy to use the remapping heuristics on the coarsest graphs of the multilevel process (rather than the final partition).

However, a question arises from this choice of procedures: how does the solver know which approach to use *a priori*, and, if it chooses the diffusive route, how does it manage the trade-off between cut-weight and data migra-

tion? (N.B. Since the scratch-remapping is two phase optimization/assignment approach, there is no possibility for a trade-off and the migration is purely a function of the partition found.)

Furthermore, consider a typical situation in which a CFD solver simulates the build up of a shock wave – initially the mesh will change frequently and so data migration may be of prime concern; however, as the solution approaches a steady state, remeshes become infrequent and so the emphasis should perhaps be on finding the very best quality partition in order to minimize the cost of repeated halo updates of solution variables. In other words, the correct trade-off is not even fixed.

1.3.1 Motivation

In this chapter we attempt to answer the question raised above by deriving a general framework which can handle both large and small changes in the partition balance, and which elegantly manages the migration/cut-weight trade-off. To motivate the ideas, we first consider some related work, and then discuss the issue of local matching.

1.3.1.1 Related work In developing the fairly simple ideas behind this chapter, we borrowed from a selection of previous work, and three papers in particular, all of which have attempted to address the trade-off between cut-weight and data migration. For example, in an early attempt, Walshaw & Berzins [28] condensed internal vertices (i.e. those at some chosen distance from subdomain borders) to form 'super-vertices', one per subdomain, and then employed the standard recursive spectral bisection (RSB) algorithm [18], on the resulting graph. Not only did this considerably reduce the computational expense of RSB, but also prevented excessive data migration since the the super-vertices, and their contents, were fixed in their home subdomain. However, it results in rather an inflexible strategy – once condensed, vertices that make up the super-vertices cannot ever migrate, even if necessary to balance the load or improve partition quality.

Another strategy for controlling the migration/cut-weight trade-off is to use local matching, but to terminate the graph coarsening early (once the graph size falls below some threshold), and then use diffusive load-balancing in conjunction with the refinement phase [22, 26]. As mentioned in §1.2.1.2, this works because the more coarsening that occurs, the more 'global' the partition quality and hence, in principle, the smaller the cut-weight and the larger the data migration. The threshold can thus serve as a parameter to manage the migration/cut-weight trade-off. However, although it works reasonably well (at least in situations where the graph has not changed too much), this parameter is rather crude and hence affords little control over the trade-off.

Perhaps most closely related to the work here, however, is the multilevel directed diffusion (MLDD) algorithm of Schloegel *et al.* [15]. They used the observation that, if only local matching is allowed, *every* vertex, both those

in the original graph and those in the coarsened graph, has a home processor/subdomain. (Recall from §1.2.1.1, local matching means that vertices are only allow to match with those in the same subdomain, and so any coarsened vertex is purely made up of vertices from one subdomain.) They then classified vertices as **clean** if they were still assigned to their home processor, or **dirty** if they had moved away. This meant they could modify the refinement algorithm, a similar variant of KL to that described in §1.2.2, to prefer the migration of dirty vertices to clean ones (since they do not increase migration any further). Furthermore they defined a cleanness factor (CF) which allowed the algorithm to control the movement of clean vertices – for example, with the CF set to a small value, clean vertices were *only* allowed to migrate if they reduced the cut-weight. However, they found that the modified algorithm only worked if the graph had not changed too much, and they also found that in certain situations the diffusive load-balancing could compromise the partition quality (hence their introduction of their wavefront diffusion scheme in [17]).

In all of these schemes, although the intent to control data migration is evident, there are still some issues that arise. Firstly, as mentioned above, it is not clear when to use the scratch-remap scheme rather than diffusive load-balancing and, indeed, when using diffusive load-balancing, it does seem that it can sometimes damage the partition quality by forcing vertices to move according to the balancing flow computed by the load-balancing algorithm (see §1.2.2). For example, if two subdomains with very different loads are adjacent, then the balancing flow may require that a large number of vertices flow across their common border. However, if that border is very short (a factor which the load-balancer does not take into account) then this flow can seriously distort the partition and hence increase cut-weight.

1.3.1.2 Local matching All three schemes mentioned above use a form of local matching. Whilst this does seem to result in lower data migration, we have noticed that sometimes it may inhibit the multilevel algorithm from doing the best job it can, since it places artificial restrictions on the multilevel coarsening.

To see this, consider the coarsenings shown in Figure 1.4, where the dashed line indicates an initial partition. Clearly, there are a number of ways that this tiny graph could be coarsened, but if only local matching is allowed, then that indicated by the rings in Figure 1.4(a) is quite likely. In particular, vertices on the subdomain borders are forced to match with each other or with internal vertices. When such a matching is used for coarsening, the graph shown in Figure 1.4(b) results (here the larger vertices now have weight 2 and the thicker line indicates an edge of weight 2). The second coarsening is more straightforward; if the heavy-edge heuristic is used (see §1.2.1.1) it is very likely that the matching shown by the single ring will occur, resulting in the coarsened graph shown in Figure 1.4(c). Unfortunately, however, neither graph in (b) or (c) is much good for finding the optimal partition (since the optimal edges to cut have been hidden by the coarsening) and so the multilevel

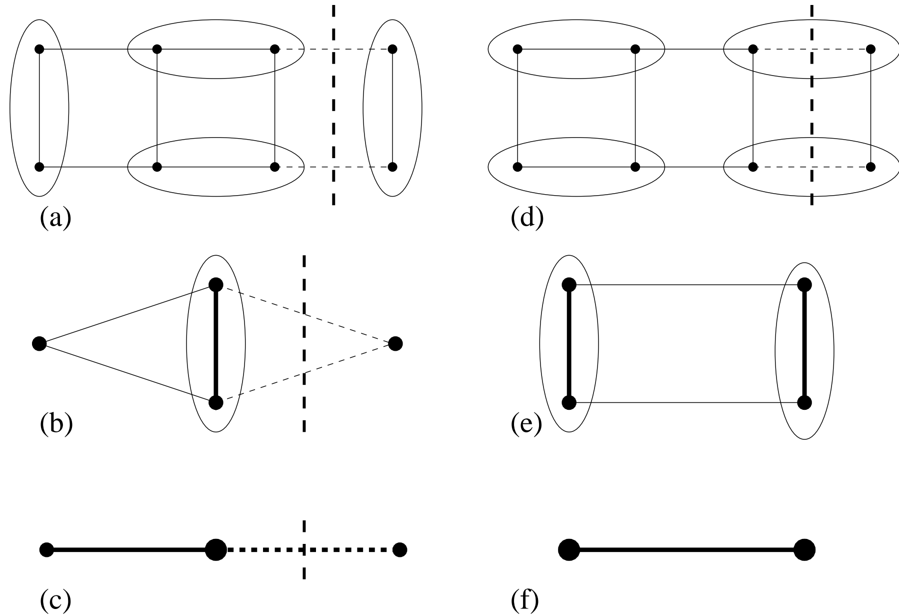


Fig. 1.4 An example partitioned graph being coarsened via local matching, (a)-(c), and non-local matching, (d)-(f).

partitioner must wait until the original graph before it can solve the problem. In other words, edges close to the subdomain borders are very likely to be collapsed early on by local matching and yet these edges are often the very ones that the partitioner is likely to want to cut.

Conversely, suppose we do not restrict the matching of vertices to others in the same subdomain and the matching shown in Figure 1.4(d) is chosen. This results in the graph in Figure 1.4(e) and indeed if this graph were coarsened using the heavy-edge heuristic, the graph in Figure 1.4(f) would inevitably result. Unlike the previous case, however, the optimal partition can now be found from the coarsest graph!

To summarize then, the local matching, although a heuristic that certainly reduces vertex migration, may actually act against the needs of the partitioner.

Of course, this is only a tiny example in which we have deliberately chosen bad and good matchings. However, although coarsening is typically a very randomized process (since, even with the heavy edge heuristic in place, the vertices are usually visited in random order), once certain initial matching choices have been made, many others are forced on the graph. For example, in Figure 1.4(d), once the vertex in the top left hand corner has matched with the next vertex along horizontally, then the vertex in the bottom left hand corner is also forced to match horizontally. Indeed, in Figure 1.4(e)

the matching shown is the *only* one which the heavy-edge heuristic would compute.

Nonetheless, it is interesting to see that the issue can be demonstrated in such a small graph, and it seems quite possible that an accumulation of many sub-optimal coarsening choices, forced by local matching, could have deleterious effects on the partition quality. This is also borne out by experimentation (see below).

1.3.2 The inertia graph

In order to overcome the issues raised in §1.3.1 we borrow ideas from the papers mentioned above. Firstly we want a consistent strategy which can be employed whether or not the graph has changed dramatically, and this mitigates against the use of diffusive load-balancing. Furthermore, we want to give the coarsening complete freedom to create the hierarchy of graphs and, in particular, allow non-local matching (i.e. the matching of vertices in different subdomains). However, we also need to control the vertex migration explicitly.

1.3.2.1 Building the graph In fact a simple solution presents itself: We first add P zero-weighted vertices to the graph, $\{\overline{v}_1, \ldots, \overline{v}_P\}$, one for each subdomain, which we refer to as subdomain vertices. We then attach every ordinary vertex, v, to its home subdomain vertex using a weighted edge, $(v, \overline{v}_p)$, where v is in subdomain S_p in the initial partition. The weight on these edges will reflect in some way how much we wish to restrict vertex migration.

We refer to this new graph, an augmented version of the original, as the **inertia graph**, $\overline{G}(\overline{V}, \overline{E})$, so called because it encourages an inertia against vertex migration. Here, the number of vertices is given by $|\overline{V}| = |V| + P$ and, since we add one edge for every existing vertex, the number of edges is given by $|\overline{E}| = |V| + |E|$.

Figure 1.5(a) shows an example of a small partitioned graph and Figure 1.5(b) the corresponding inertia graph. Figures 1.5(c) & 1.5(d) then show the first and second coarsenings of this graph. A number of non-local matchings, indicated by a coarsened vertex having edges to two (or potentially more) subdomain vertices, can be seen in both of these figures, for example centre top of Figure 1.5(d).

An important point is that the inertia graph represents exactly the same problem as the original graph in terms of load-balance, since the subdomain vertices are zero weighted. However, every vertex that migrates away from its original subdomain results in an additional contribution to the partition cost, the impact of which is determined by the ratio of ordinary edge weights to inertial edge weights.

We can then use exactly the same multilevel partitioning techniques on the graph as described in Section 1.2, provided we ensure that none of the subdo-

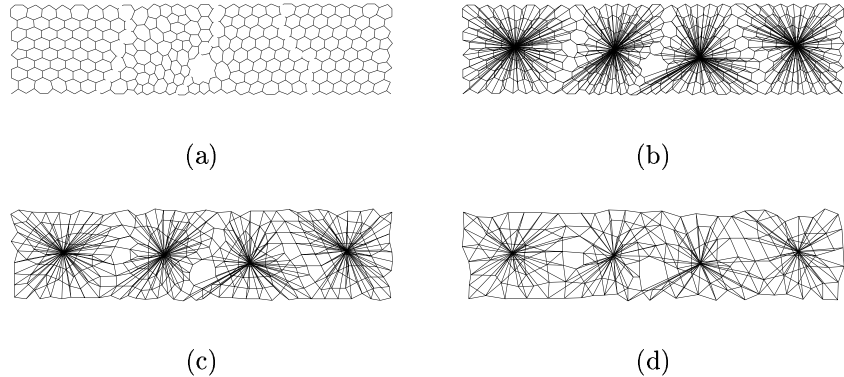


Fig. 1.5 An example graph showing (a) the initial partition, (b) the corresponding inertia graph, (c) the first coarsening, and (d) the second coarsening.

main vertices are ever allowed to migrate. We also want to leave the matching essentially unchanged, so we never allow the ordinary vertices to match with subdomain vertices. These two features are achieved by implementing the subdomain vertices as **fixed** vertices which are not allowed to migrate, nor to match with other vertices. (In fact, fixed vertices were already implemented in JOSTLE as part of an approach to the multiphase partitioning problem [27].)

1.3.2.2 Weighting the edges The question then arises, what weight should be put on the new inertial edges between vertices and subdomain vertices? We want the weights to control the vertex migration, and indeed, as mentioned in the introduction to Section 1.3, the priorities may change as the solver progresses. It is therefore more useful to think about the ratio between ordinary and inertial weights.

Consider first the simplest case, where the original graph edges are unweighted, i.e. they all have a weight of 1. If the ratio is 1:3 then we could simply set every inertial weight to 3. Conversely, if we required a ratio of 3:1, that suggests setting the weight of the inertial edges to $1/3$. However, non-integer weights complicate the partition refinement algorithms (although they can be achieved – see [25]) and so instead we could set every inertia weight to 1 and then adjust every ordinary edge by giving it an additional weight of 2. (Note that adding a constant weight to every ordinary edge does not in itself change the partitioning problem, since the change in the cost function of moving a vertex from one subdomain to another is calculated as a relative quantity.)

In our initial experiments we then found that the results for different ratios were rather dependent on the type of graph being employed. Consider, for

example, the case where the weights on the inertial edges are all set to 1. A 2D dual graph which represents a mesh formed of triangles (see §1.1.2) will have a maximum vertex degree of 3 and so an inertial edge of weight 1 has quite a large effect. Conversely, in a 3D nodal graph representing a mesh formed of tetrahedra, a vertex might have a degree of 20 or 30 and so the effect of the inertial edge is negligible.

To counteract this, we calculated the average vertex edge weight $\overline{e} = \|E\|/|V|$, the average weight of edges incident on each vertex (rounded to the nearest integer value). Then, if we require a ratio of w_e:w_i between ordinary and inertial weights, where, without much loss of generality w_e and w_i are positive integers, we simply set the inertia edges to have the weight $w_i \times \overline{e}$ and add $(w_e - 1)$ to the ordinary edge weights. This also covers the case of graphs with weighted edges.

To give two examples from the test graphs in §1.4.1, brack2, a 3D nodal graph, has 62,631 vertices and 366,559 edges, each of weight 1, so the average vertex edge weight is 5.85, rounded up to $\overline{e} = 6$. Thus for a ratio of 1:1 we set the inertial edge weights to 6 and leave the ordinary edge weights at 1; whilst for a ratio of 5:1 we set the inertial edge weights to 6 and add $4 = (5-1)$ to the ordinary edge weights. Meanwhile, mesh100, a 3D dual graph, has 103,081 vertices and 200,976 edges, each of weight 1, so the average vertex edge weight is 1.94, rounded up to $\overline{e} = 2$. Thus for a ratio of 1:1 we set the inertial edge weights to 2 and leave the ordinary edge weights at 1; whilst for a ratio of 5:1 we set the inertial edge weights to 2 and again add $4 = (5 - 1)$ to the ordinary edge weights. With this scheme in place, experimentation indicates that a given ratio produces similar effects across a wide range of graph types.

1.3.2.3 Discussion As mentioned, the partition inertia framework borrows from existing ideas and the strategy of modifying the graph and then using standard partitioning techniques has been successfully employed previously, e.g. [25, 27]. Indeed, in a very similar vein, both Hendrickson & Leland [8], and Pellegrini & Roman [14], used additional graph vertices, essentially representing processors/subdomains, although in these cases the aim was to enhance data locality when mapping onto parallel machines with non-uniform interconnection architectures (e.g. a grid of processors or a meta-computer).

Superficially, though, the partition inertia framework is most similar to the multilevel directed diffusion (MLDD) algorithm of Schloegel *et al.* [15], as described in §1.3.1.1, with the ratio of inertia edge weights to ordinary edge weights (the parameter that controls the trade-off of cut-weight and migration), mirroring the cleanness factor.

However, it differs in three important aspects. Firstly, the MLDD scheme relies on local matching – indeed without the local matching there is no means to determine whether a coarsened vertex is 'clean' or 'dirty'. Secondly, MLDD uses diffusion in the coarsest graphs to balance the load, whereas the partition inertia framework uses a more general multilevel partitioning scheme which is not diffusive in nature (indeed, since non-local matching is allowed, the ini-

tial partition is completely ignored throughout the coarsening and the coarsest graph is likely to be better balanced than the initial one since non-local matching may result in migrations which can be chosen to improve the balance). It is true that a balancing flow, calculated via a diffusive algorithm, is incorporated into JOSTLE's refinement algorithm to adjust partitions which become imbalanced, but this is simply a standard tool in JOSTLE's armoury which is frequently never used. Thirdly, and most importantly, the MLDD scheme is implemented by restricting the matching and then modifying the refinement algorithm, whereas partition inertia is a much more general framework which works by modifying the input graph and, in principle, could use any high quality partitioning scheme (with the minor requirement that the P fixed vertices never migrate from their home subdomain).

1.3.3 Setting the ratio: a self-adaptive framework

Of course, it is not easy to judge what ratio to choose for the inertial and ordinary weights. Even with extensive experimentation it would be hard to predict for any given problem. In addition, as has been mentioned, the relative weighting is quite likely to change throughout the simulation.

As a result we propose the following simple scheme. Initially the ratio is set to something reasonable, chosen empirically. In the experiments below, Section 1.4, an appropriate ratio seems to be around 5:1 in favor of the ordinary edge weights. However, this is dependent on the characteristics of the simulation.

Subsequently, all that is required from the solver is that it monitor the parallel overhead due to halo updates of solution variables and estimate the data migration time at each remesh. It then simply passes an argument to the partitioner, expressing whether halo updates take more, less or about the same time. If the halo updates are taking longer, the ratio is increased (e.g. 6:1, 7:1, ...) in favor of ordinary edge weights, with the aim of improving cut-weight. Conversely, if the data migration time is longer, the partitioner decreases the ratio (e.g. 4:1, 3:1, ...) and if this trend continues beyond 1:1 it is increased in the other direction (e.g. 2:1, 1:1, 1:2, 1:3, ...). Not only is this extremely simple, it requires little monitoring on the part of the solver.

Note that, whilst this idea is very straightforward and could easily have been implemented in previous work (e.g. [22]), where the trade-off was controlled by the coarsening threshold, in fact, the problem with that was that the threshold parameter is rather a crude one which gives relatively little control over migration. However, the experimental results indicate that with variable partition inertia, the correlation between the parameter setting (the edge weight ratio) is much more stable and hence provides a much more robust method for managing the trade-off.

1.4 EXPERIMENTAL RESULTS

As discussed in Section 1.2, the software tool written at Greenwich and which we use to test the variable partition inertia concept is known as JOSTLE. It is written in C and can run in both serial and parallel, although here we test it in serial.

In the following experiments we use two metrics to measure the performance of the algorithms – the cut-weight percentage, $\|E_c\|/\|E\|$, and the percentage of vertices which need to be migrated, $|V_m|/|V|$. We do not give run times since, essentially, the partition inertia scheme is running exactly the same algorithms as JOSTLE-MS. It is true that the input graph is extended by having subdomain vertices and inertial edges, but since these are excluded from most calculations, the run times are broadly similar.

1.4.1 Sample results

We first provide some sample results from fairly well known mesh-based graphs[1], often used for benchmarking partitioners. For each of the three meshes we have generated three high quality, but imbalanced, partitions calculated by JOSTLE with the permitted imbalance set to approximately 25%, 50% and 100% (although JOSTLE was never written to hit these relaxed constraints exactly, so some of the imbalances vary a little). We then test each configuration using these partitions as a starting point.

Firstly, to compare with our previous work and the algorithms discussed in [22], it is run in three previous configurations, dynamic (JOSTLE-D), multilevel-dynamic (JOSTLE-MD) and multilevel-static (JOSTLE-MS). The two dynamic configurations primarily use diffusive load-balancing: JOSTLE-D, reads in the existing partition and uses the single-level refinement algorithm outlined in §1.2.2 to balance and refine the partition, whilst JOSTLE-MD, uses the same procedure but incorporated within the multilevel framework (§1.2.1) to improve the partition quality. JOSTLE-MD also uses local matching and, as discussed in §1.2.1.2, a coarsening threshold to control the trade-off between cut-weight and vertex migration (set to $20P$ for all the experiments below). Finally, the static version, JOSTLE-MS, uses the same core algorithms, but ignores the initial partition entirely and hence provides a set of control results.

To demonstrate the variable partition inertia (VPI) framework, we employ three different settings. Recall from §1.3.2 that the paradigm involves modifying the input graph by attaching an inertial edge between every vertex and its home subdomain. The graph edges are then weighted according to a ratio, $w_e{:}w_i$, which expresses the difference between ordinary and inertial edge weights.

[1] available from `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition`

After some preliminary experimentation, we chose the ratios 10:1, 5:1 and 1:1 for the results presented here. A ratio of 1:1 gives approximately equal weighting to inertial and ordinary edges and hence puts a lot of emphasis on minimising vertex migration. Meanwhile, a ratio of 10:1 makes the inertial edges very much weaker than any ordinary edge and hence puts the emphasis on minimising cut-weight (although still preventing too much migration).

Table 1.1 Migration results for the sample graphs

	JOSTLE			JOSTLE(VPI)		
graph/imbalance	MS	MD	D	10:1	5:1	1:1
4elt/1.25	96.5%	11.6%	10.0%	8.8%	7.4%	7.7%
4elt/1.49	96.8%	28.1%	27.7%	22.1%	23.9%	19.7%
4elt/1.98	92.5%	47.1%	44.5%	36.8%	34.7%	34.8%
brack2/1.22	95.9%	14.7%	14.8%	16.2%	11.3%	9.8%
brack2/1.55	96.7%	29.6%	30.8%	26.7%	24.3%	21.5%
brack2/1.99	93.0%	41.5%	44.3%	43.8%	37.2%	36.7%
mesh100/1.30	87.4%	16.2%	17.0%	15.7%	14.4%	13.5%
mesh100/1.50	96.9%	48.1%	43.3%	34.9%	28.0%	22.8%
mesh100/1.89	92.6%	50.9%	49.3%	44.8%	42.7%	38.0%
average	94.3%	32.0%	31.3%	27.8%	24.9%	22.7%
local				28.0%	24.2%	23.3%
IML				25.0%	22.4%	22.1%

The migration and cut-weight results are presented in Tables 1.1 & 1.2 respectively. The averages for each partitioning scheme over the nine results are shown at the bottom and it is instructive to focus on these. Firstly, and as should be expected, the trend lines of the two different approaches show how migration can be traded-off against cut-weight. Because JOSTLE-MS takes no account of the existing partition it results in a vast amount of data migration (94.3%). However, and as shown in [22], the dynamic diffusive partitioners are able to make a considerable difference and reduce this to just over 30%. The cut-weight results increase accordingly though (2.77% up to 3.27%) and if less migration takes place (JOSTLE-D), the partitioner cannot refine as well, and the cut-weight driven up. Interestingly, however, and as suggested in [15], with these very unbalanced initial partitions the diffusive partitioners may not be the method of choice and JOSTLE-D does little better in terms of migration than its multilevel counterpart, JOSTLE-MD.

The same trends are in evidence with the VPI results – as the edge weight ratio decreases from 10:1 to parity (thus putting increasing emphasis on minimising migration), the migration certainly decreases, but at the cost of increasing cut-weight. However, comparing the two, JOSTLE(VPI) does much better than the diffusive partitioners. For example, with the ratio 10:1, JOS-

Table 1.2 Cut-weight results for the sample graphs

graph/imbalance	JOSTLE			JOSTLE(VPI)		
	MS	MD	D	10:1	5:1	1:1
4elt/1.25	2.32%	2.21%	2.31%	2.25%	2.30%	2.42%
4elt/1.49	2.32%	2.26%	2.48%	2.43%	2.52%	3.02%
4elt/1.98	2.32%	2.35%	2.71%	2.58%	2.60%	3.92%
brack2/1.22	3.59%	3.60%	3.84%	3.54%	3.57%	3.82%
brack2/1.55	3.59%	3.85%	3.97%	3.57%	3.72%	4.45%
brack2/1.99	3.59%	3.87%	4.41%	3.60%	3.74%	3.89%
mesh100/1.30	2.38%	2.42%	2.73%	2.36%	2.41%	2.97%
mesh100/1.50	2.38%	2.61%	3.65%	2.62%	2.83%	3.41%
mesh100/1.89	2.38%	2.33%	3.29%	2.58%	2.78%	3.37%
average	2.77%	2.83%	3.27%	2.84%	2.94%	3.47%
local				2.93%	3.00%	3.48%
IML				2.70%	2.80%	3.24%

TLE(VPI) has almost identical cut-weight to JOSTLE-MD (2.84% against 2.83%) but considerably better migration (27.8% against 32.0%) and with the ratio 1:1, JOSTLE(VPI) even reduces migration down to 22.7%. Perhaps most importantly, however, the VPI framework appears to offer much finer control over the migration/cut-weight trade-off.

At the bottom of the tables we also present averages for two other flavors of the VPI scheme. The row marked 'local' uses local matching (and hence should produce very similar results to the MLDD scheme, as discussed in §1.3.2.3). However, as suggested in §1.3.1.2, it does seem that the local matching results in worse partition quality (2.93%-3.48% against 2.84%-3.47%) without improving the migration (28.0%-23.2% against 27.8%-22.7%).

Finally, the row marked IML uses iterated multilevel refinement (§1.2.4), repeatedly coarsening and uncoarsening to find better results. Although not a serious contender in a time-critical application such as the repartitioning of parallel adaptive meshes, it does indicate very well the flexibility of the VPI framework. Indeed, since we have just modified the input graph, we can, in principle, use any partitioning scheme to attack the problem, and here, by using a very high quality one, albeit much more time-consuming, we can find even better results. For example, the ratio 10:1 used in iterated JOSTLE(VPI) finds the best cut weight results (2.70%) in combination with very good migration results (25.0%).

1.4.2 Laplace solver adaptive mesh results

To give a fuller flavor of the VPI framework we carried out further experiments on two sets of adaptive unstructured meshes.

The first set of test meshes have been taken from an example contained in the DIME (distributed irregular mesh environment) software package [29]. This particular application solves Laplace's equation with Dirichelet boundary conditions on a square domain with an S-shaped hole and using a triangular finite element discretisation. The problem is repeatedly solved by Jacobi iteration, refined based on this solution and then load-balanced. Similar sets of meshes have previously been used for testing repartitioning algorithms [17, 22, 26] and details of DIME can be found in [30].

For the test configurations, the initial mesh is partitioned with the static version – JOSTLE-MS. Subsequently at each refinement, the existing partition is interpolated onto the new mesh using the techniques described in [28] (essentially, new elements are owned by the processor which owns their parent) and the new partition is then refined.

As for the sample graphs, the experiments have been run using JOSTLE-MS as a control, two example diffusive partitioners, JOSTLE-MD and JOSTLE-D, and a number of partition inertia variants. In particular, we use the same three edge weight ratio settings as above, 10:1, 5:1 and 1:1, and two variable schemes which we discuss below. Note that we now distinguish between the schemes where the edge weight ratios are fixed, JOSTLE(PI), and the variable ones, JOSTLE(VPI).

Table 1.3 Migration results for the Laplace graphs

	JOSTLE			JOSTLE(PI)			JOSTLE(VPI)	
G	MS	MD	D	10:1	5:1	1:1	10↓:1	1↑:1
1	90.3%	17.3%	10.6%	23.2%	14.8%	9.2%	23.2%	9.2%
2	93.6%	18.5%	12.9%	20.4%	15.7%	10.4%	17.4%	11.4%
3	98.2%	13.3%	5.8%	10.7%	9.2%	6.7%	12.2%	10.2%
4	86.5%	9.3%	6.7%	12.8%	8.6%	5.6%	7.4%	8.7%
5	97.8%	6.4%	2.5%	8.6%	5.0%	1.5%	5.7%	4.8%
6	98.2%	5.5%	3.1%	9.3%	4.9%	1.6%	4.1%	6.3%
7	92.7%	6.8%	1.9%	3.7%	4.0%	0.6%	2.1%	0.0%
8	100.0%	4.0%	1.3%	3.7%	2.1%	0.6%	2.1%	3.2%
9	96.6%	4.9%	1.3%	3.2%	2.7%	0.7%	0.5%	3.9%
avg	94.9%	9.6%	5.1%	10.6%	7.5%	4.1%	8.3%	6.4%
local				5.9%	5.4%	4.2%	5.4%	5.5%
IML				7.6%	5.9%	4.0%	7.2%	5.4%

Table 1.4 Cut-weight results for the Laplace graphs

	JOSTLE			JOSTLE(PI)			JOSTLE(VPI)	
G	MS	MD	D	10:1	5:1	1:1	10↓:1	1↑:1
1	5.19%	5.73%	5.86%	5.77%	5.86%	6.27%	5.77%	6.27%
2	4.04%	4.59%	5.34%	4.95%	4.88%	5.92%	4.72%	5.14%
3	3.57%	3.74%	4.86%	3.95%	3.98%	5.10%	3.91%	4.26%
4	3.35%	3.13%	4.12%	3.30%	3.49%	4.37%	3.13%	3.51%
5	2.86%	2.40%	3.33%	2.72%	2.78%	3.50%	2.77%	2.85%
6	2.31%	2.00%	3.00%	2.24%	2.45%	3.07%	2.31%	2.48%
7	1.94%	1.74%	2.53%	1.91%	1.96%	2.55%	1.96%	2.20%
8	1.71%	1.48%	2.19%	1.59%	1.69%	2.25%	1.70%	1.73%
9	1.40%	1.34%	1.92%	1.37%	1.46%	1.95%	1.51%	1.47%
avg	2.93%	2.91%	3.68%	3.09%	3.17%	3.88%	3.09%	3.32%
local				3.17%	3.52%	3.91%	3.30%	3.43%
IML				2.96%	3.06%	3.75%	3.04%	3.30%

The migration and cut-weight results are presented in Tables 1.3 & 1.4 respectively, and, once again, looking at the averages, the trend lines are in the same directions. From left to right, as the vertex migration decreases, the cut-weight increases. In fact, for this set of results, the best average cut-weight is found by JOSTLE-MD and the VPI scheme is unable to match this. However JOSTLE-MD results in 9.6% migration and if vertex migration is the most important factor, then the VPI scheme 1:1 manages to find the minimum value of 4.1%.

As mentioned in the introduction to Section 1.3, however, as the simulation progresses, the needs of the solver may change with respect to the migration/cut-weight trade-off. To test this, we looked at the trend of the results and tried two schemes, indicated by 10↓:1 and 1↑:1, in which the edge weight ratio changes with each mesh. For example, if we look at the cut-weight results for individual meshes, we can see that they decrease from between 5-6% down to 1-2% as the simulation progresses. In other words, the cut-weight plays a more important role at the beginning of the simulation than at the end. This suggests a VPI scheme that does the same and so we tested the 10↓:1 scheme where the ratio starts at 10:1 for the first mesh and then decreases towards parity with every successive mesh (i.e. 10:1, 9:1, 8:1, ...). The average results indicate the success of this strategy – when compared with 10:1, it achieves the same cut-weight (3.09%) but improves on the migration (8.3% against 10.6%).

Similarly, we looked at the migration, which also plays a more important role at the beginning of the simulation than at the end, and tried to match this with the scheme 1↑:1 (with ratios of 1:1, 2:1, 3:1, ...). Once again, this

improved on the 1:1 scheme and fits well into the trade-off curve of migration versus cut-weight.

In summary, these two variable schemes give some indication of how the VPI framework can track the needs of the solver. In this particular case, it is not clear which of the two schemes to use, but this would depend heavily on the relative costs of data migration and halo updates of solution variables.

Finally, we used these graphs to test local matching and iterated multilevel refinement in combination with the VPI framework (the two rows at the bottom of the table). Once again, the results demonstrate that local matching can harm partition quality, and that the VPI framework is not dependent on a particular partitioner – if given more time for searching, such as with the iterated multilevel algorithm, it can find considerably better results (e.g. iterated JOSTLE(PI) with 10:1 ratio).

Overall then, for this set of test meshes the results are inconclusive in the comparison of diffusive repartitioners against VPI based schemes. However, the VPI framework does give a lot more control over the trade-off, particularly when we employ the variable version.

1.4.3 CFD adaptive mesh results

Our second set of test meshes come from a CFD simulation in which a shock wave builds up rapidly. Commercial sensitivity prevents us giving details of the meshes, but they are of medium size and consist of 2D triangular elements.

Table 1.5 Migration results for the CFD graphs

	JOSTLE			JOSTLE(PI)			JOSTLE(VPI)	
G	MS	MD	D	10:1	5:1	1:1	10↓:1	1↑:1
1	73.4%	7.0%	0.5%	3.8%	3.0%	0.5%	3.8%	0.5%
2	46.7%	9.1%	5.9%	4.9%	5.5%	4.0%	4.8%	3.8%
3	53.3%	12.5%	7.0%	9.7%	8.5%	5.9%	12.2%	7.6%
4	53.7%	16.0%	4.6%	7.8%	5.6%	3.5%	6.9%	8.3%
5	93.5%	18.4%	4.8%	21.0%	17.2%	7.4%	12.1%	12.2%
6	57.4%	19.0%	8.8%	12.3%	14.1%	5.9%	9.8%	9.9%
7	68.6%	24.2%	14.1%	22.6%	15.6%	12.9%	18.1%	18.7%
8	55.2%	7.4%	6.5%	8.5%	6.5%	6.1%	9.3%	9.1%
9	91.5%	7.4%	5.6%	6.8%	5.8%	2.4%	4.3%	8.3%
avg	65.9%	13.5%	6.4%	10.8%	9.1%	5.4%	9.0%	8.7%
local				14.4%	7.1%	5.3%	7.5%	16.9%
IML				7.8%	7.3%	5.4%	7.2%	7.4%

As above, we use JOSTLE-MS as a control, two example diffusive partitioners, JOSTLE-MD and JOSTLE-D, and the same partition inertia variants as

Table 1.6 Cut-weight results for the CFD graphs

	JOSTLE			JOSTLE(PI)			JOSTLE(VPI)	
G	MS	MD	D	10:1	5:1	1:1	10↓:1	1↑:1
1	3.24%	3.16%	3.09%	3.01%	3.01%	3.09%	3.01%	3.09%
2	2.86%	2.86%	2.86%	3.10%	3.10%	3.10%	3.19%	3.27%
3	3.01%	3.01%	5.02%	3.18%	3.60%	4.10%	3.51%	3.43%
4	2.90%	3.49%	4.86%	3.49%	3.49%	4.00%	2.98%	3.49%
5	3.10%	3.63%	7.79%	3.19%	4.60%	5.40%	4.34%	4.16%
6	3.28%	3.56%	6.10%	3.00%	3.47%	5.72%	3.66%	4.03%
7	3.02%	2.87%	7.92%	3.09%	2.94%	5.51%	3.47%	3.32%
8	2.76%	2.76%	6.49%	2.99%	3.21%	4.63%	3.06%	3.21%
9	3.86%	2.78%	6.87%	2.93%	3.24%	5.95%	3.17%	3.01%
avg	3.11%	3.13%	5.67%	3.11%	3.41%	4.61%	3.38%	3.45%
local				3.51%	3.33%	5.23%	3.53%	3.39%
IML				2.95%	3.06%	4.64%	3.08%	3.20%

above, 10:1, 5:1, 1:1, and the variable schemes 10↓:1 and 1↑:1. The migration and cut-weight results are presented in Tables 1.5 & 1.6 respectively.

As before, the average trend lines are in the right directions; from left to right, as the vertex migration decreases, the cut-weight increases. However, for this set of results the VPI strategy finds the best results. For example, the 10:1 weighting finds the best cut-weight (matched by JOSTLE-MS), better than JOSTLE-MD in this case, and also manages to improve on the vertex migration, as compared with JOSTLE-MD. Similarly, the 1:1 weighting beats JOSTLE-D in both cut-weight and migration.

Considering the variable schemes, the choice of whether to increase or decrease the edge weight ratio is not as clear as in §1.4.2; the cut-weight seems more or less constant throughout the simulation, and the migration has no obvious trend. Nevertheless, the 10↓:1 scheme improves slightly on the 5:1 weighting, whilst the 1↑:1 provides very similar results.

Finally, and once again, the local matching and iterated multilevel results support the same results as before, that local matching can harm the partition quality and that the VPI framework is both robust and flexible.

1.5 SUMMARY AND FUTURE WORK

We have presented a new framework, variable partition inertia (VPI), for the repartitioning of adaptive unstructured meshes. It is simple to implement, since it merely involves manipulation of the input graph (plus the minor requirement that the partitioner can handle fixed vertices). In principle, it

can therefore be used with any partitioner and the results indicate that it is robust and flexible. Most importantly, VPI seems to afford much greater control over the repartitioning problem than diffusive load-balancing and the edge weighting ratio gives a powerful parameter for managing the trade-off between cut-weight and migration.

We have also indicated a scheme for varying the edge weight ratio, based solely on run-time instrumentation within the solver.

As part of the work, we have also demonstrated, both by example and empirically, that the well-accepted local matching can damage partition quality (albeit only a little).

In the future, it would be of interest to validate the variable edge weight scheme by running tests alongside a genuine parallel adaptive simulation. It would also be instructive to test the VPI framework against the scratch-remapping schemes (see Section 1.3), currently held to be the best approach when the mesh has undergone dramatic changes.

REFERENCES

1. V. Aravinthan, S. P. Johnson, K. McManus, C. Walshaw, and M. Cross. Dynamic Load Balancing for Multi-Physical Modelling using Unstructured Meshes. In C.-H. Lai *et al.*, editor, *Proc. 11th Intl. Conf. Domain Decomposition Methods, Greenwich, UK, 1998*, pages 380–387. DDM.org, www.ddm.org, 1999.

2. S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

3. R. Biswas and L. Oliker. Experiments with Repartitioning and Load Balancing Adaptive Meshes. Tech. Rep. NAS-97-021, NAS, NASA Ames, Moffet Field, CA, 1997.

4. T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, Philadelphia, 1993.

5. C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181. IEEE, Piscataway, NJ, 1982.

6. B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. *Comput. Methods Appl. Mech. Engrg.*, 184(2-4):485–500, 2000.

7. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95, San Diego*. ACM Press, New York, 1995.

8. B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing Data Locality by Using Terminal Propagation. In *Proc. 29th Hawaii Intl. Conf. System Science*, 1996.

9. Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice & Experience*, 10(6):467–483, 1998.

10. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

11. G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

12. B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Syst. Tech. J.*, 49:291–308, 1970.

13. K. McManus, C. Walshaw, M. Cross, P. F. Leggett, and S. P. Johnson. Evaluation of the JOSTLE mesh partitioning code for practical multiphysics applications. In A. Ecer *et al.*, editor, *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pages 673–680. Elsevier, Amsterdam, 1996. (Proc. Parallel CFD'95, Pasadena, 1995).

14. F. Pellegrini and J. Roman. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I, 33405 TALENCE, France, 1996.

15. K. Schloegel, G. Karypis, and V. Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

16. K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. TR 98-034, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1998.

17. K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, 2001.

18. H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems Engrg.*, 2:135–148, 1991.

19. C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. *Annals Oper. Res.*, 131:325–372, 2004.

20. C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000.

21. C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000.

22. C. Walshaw and M. Cross. Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes. In B. H. V. Topping, editor, *Computational Mechanics Using High Performance Computing*, pages 79–94. Saxe-Coburg Publications, Stirling, 2002. (Invited Chapter, Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999).

23. C. Walshaw and M. Cross. Parallel Mesh Partitioning on Distributed Memory Systems. In B. H. V. Topping, editor, *Computational Mechanics Using High Performance Computing*, pages 59–78. Saxe-Coburg Publications, Stirling, 2002. (Invited Chapter, Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999).

24. C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In C.-H. Lai and F. Magoules, editors, *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Civil-Comp Ltd., 2007. (Invited chapter – to appear).

25. C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. *Intl. J. High Performance Comput. Appl.*, 13(4):334–353, 1999.

26. C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

27. C. Walshaw, M. Cross, and K. McManus. Multiphase Mesh Partitioning. *Appl. Math. Modelling*, 25(2):123–140, 2000.

28. C. H. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice & Experience*, 7(1):17–28, 1995.

29. R. D. Williams. DIME: Distributed Irregular Mesh Environment. Caltech Concurrent Computation Report C3P 861, 1990.

30. R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice & Experience*, 3:457–481, 1991.