ELSEVIER

Applied Mathematical Modelling 25 (2000) 123–140

APPLIED MATHEMATICAL MODELLING

www.elsevier.nl/locate/apm

Multiphase mesh partitioning

C. Walshaw *, M. Cross, K. McManus

School of Computing and Mathematical Sciences, University of Greenwich, Park Row, Greenwich, London SE10 9LS, UK

Received 1 June 2000; accepted 1 August 2000

Abstract

We consider the load-balancing problems which arise from parallel scientific codes containing multiple computational phases, or loops over subsets of the data, which are separated by global synchronisation points. We motivate, derive and describe the implementation of an approach which we refer to as the multiphase mesh partitioning strategy to address such issues. The technique is tested on several examples of meshes, both real and artificial, containing multiple computational phases and it is demonstrated that our method can achieve high quality partitions where a standard mesh partitioning approach fails. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Graph-partitioning; Mesh-partitioning; Load-balancing; Parallel multiphysics

1. Introduction

The need for mesh partitioning arises naturally in many finite element (FE) and finite volume (FV) computational mechanics (CM) applications. Meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large to fit onto serial computers, either because of memory limitations or computational demands, or both. Distributing the mesh across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as mesh partitioning. It is well known that this problem is NP-complete, so in recent years much attention has been focused on developing heuristic methods, many of which are based on a graph corresponding to the communication requirements of the mesh, e.g. [12].

1.1. Multiphase partitioning – motivation

Typically, the load-balance constraint – that the computational load is evenly balanced – is simply satisfied by ensuring that each processor has an approximately equal share of the mesh entities (e.g. the mesh elements, such as triangles or tetrahedra, or the mesh nodes). Even in the

* Corresponding author. Tel.: +44-20-8331-8142; fax: +44-20-8331-8665.
E-mail address: c.walshaw@gre.ac.uk (C. Walshaw).

PII: S 0 3 0 7 - 9 0 4 X (0 0) 0 0 0 4 1 - X

case where different mesh entities require different computational solution time (e.g. boundary nodes and internal nodes) the balancing problem can still be addressed by weighting the corresponding graph vertices and distributing the graph weight equally. Unfortunately, for some real applications the processor load can also depend on many other factors such as data access patterns; since these are a function of the final partition, it is not possible to estimate such costs a priori and we do not address this issue here.

We therefore consider only those applications for which a reasonably accurate weighting of the graph, related to computational cost, can be realised. However even for such applications, as increasingly complex solution methods are developed, there is a class of solvers for which such simple models of computational cost break down. Consider the example shown in Fig. 1(a) with a partition for two processors indicated by dotted line. This partition might normally be considered of good quality but for the solution algorithm in Fig. 1(b) it is completely unsuitable. As Fig. 1(c) shows, during the fluid/flow phase of the calculation, processor 1 has relatively little work to do and indeed during the solid/stress phase processor 0 has no work at all. Furthermore, processor 1 is not able to start the solid/stress calculation until the fluid/flow part has terminated because of the global convergence check, a global synchronisation point (when all the processors communicate as a group).

In fact it is these *multiple loops* over subsets of the mesh entities interspersed by global communications that characterise this modified mesh partitioning problem. If, for example, all the loops in Fig. 1(b) were over all the mesh entities (as sometimes happens in codes of this nature when variables are set to zero in regions where a given phenomenon does not occur, e.g. flow in a solid) such balancing problems would not arise. Similarly, if in Fig. 1(b) there were no global convergence checks, so that a processor could commence on the stress solution immediately after the flow solution had converged locally, the problem would be removed, although the flow and stress regions might need to be weighted differently. In the simple example in Fig. 1 an obvious (and relatively good) load-balancing strategy, therefore, is simply to partition each region (i.e. liquid and solid) of the domain separately so that each processor has an equal number of entities

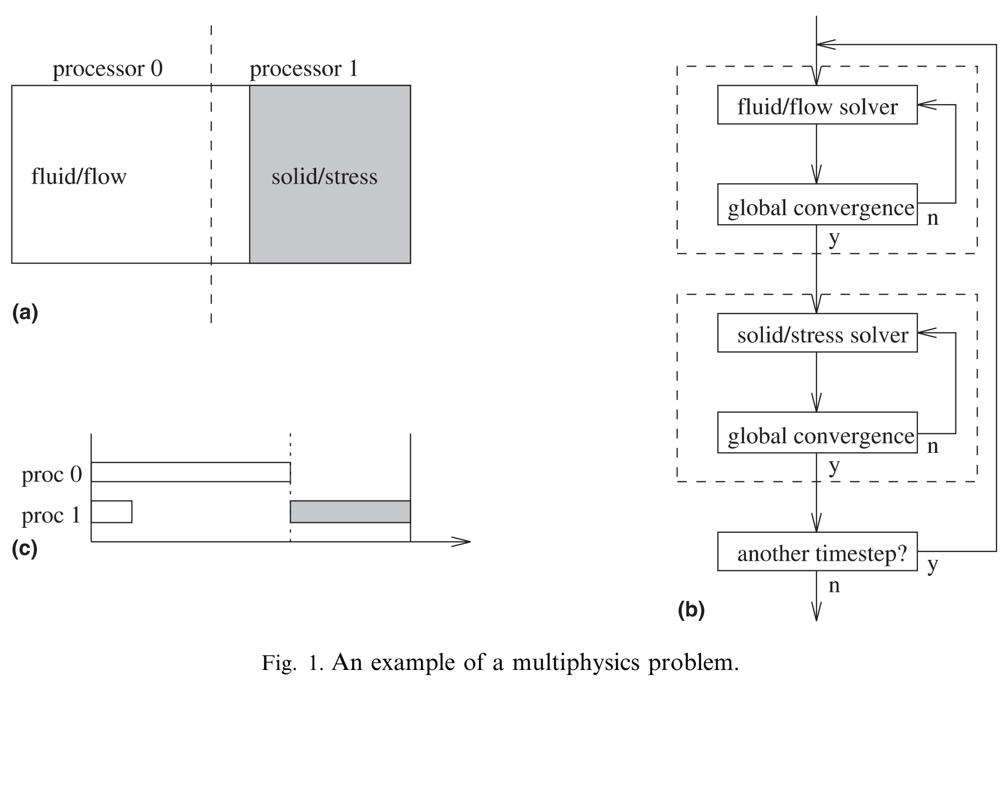

Fig. 1. An example of a multiphysics problem.

from each region. However, in more complex examples, for example where the regions relating to different computational phases overlap, this may no longer provide a good solution and a more advanced strategy is required.

We refer to this modified mesh partitioning problem as the multiphase mesh partitioning problem (MMPP) because the underlying solver has multiple distinct computational subphases, each of which must be balanced separately. Typically MMPPs arise from multiphysics or multiphase modelling (e.g. [22,23]) where different parts of the computational domain exhibit different physical behaviour and/or material properties. They can also arise in contact–impact modelling, e.g. [20], which usually involves the solution of localised stress–strain finite element calculations over the entire mesh together with a much more complex contact–impact detection phase over areas of possible penetration.

1.2. Overview

In this paper, we discuss strategies for dealing with MMPPs, primarily by extending existing single-phase mesh partitioning algorithms. A particularly popular and successful class of algorithms which address the standard or single-phase mesh partitioning problem are known as multilevel algorithms. They usually combine a graph contraction algorithm which creates a series of progressively smaller and coarser graphs together with a local optimisation method which, starting with the coarsest graph, refines the partition at each graph level. In Section 2 we outline such an algorithm and discuss the salient features. We aim to address the MMPP by using this multilevel algorithm as a 'black box' solver, partitioning the problem phase by phase, based on the partitions of the previous phases. The details of this approach are described in Section 3, in particular the necessary vertex classification scheme (Section 3.1), an overview of the strategy (Section 3.2) and modifications to the multilevel algorithm (Section 3.3). Also, note that although we describe a serial version of the multilevel algorithm, the same strategy can be used to enable parallel solution of the MMPP and in Section 3.4 we discuss a parallel implementation. Related work is discussed in Section 3.6. In Section 4 we present results for the techniques on a number of both artificial and genuine (drawn from industrial simulation) MMPPs. Finally, in Section 5 we discuss the work, present some conclusions and list some suggestions for further research.

The principal innovation described in this paper is the multiphase mesh partitioning strategy, its motivation, derivation and implementation.

2. Multilevel mesh partitioning

In this section, we discuss the single-phase mesh partitioning problem (the classical mesh partitioning problem) and outline our multilevel algorithm, described in [29], for addressing it. The modifications to the algorithm for use in the multiphase partitioning problem are deferred to Section 3.3 following the discussion of the multiphase mesh partitioning paradigm.

2.1. 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. The graph vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph), a combination of both (the full or combined graph) or some special purpose representation to model 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 (although it is often the case that vertices and edges are given unit weights, $|v| = 1$ for all $v \in V$ and $|e| = 1$ for all $e \in E$). 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 partition π induces a *subdomain graph* on G which we shall refer to as $G_\pi = G_\pi(S, L)$. There is an edge or link (S_p, S_q) in L if there are adjacent vertices $v_1, v_2 \in V$ (i.e. there is an edge $(v_1, v_2 \in E)$ and $v_1 \in S_p$ and $v_2 \in S_q$ and the weight of a subdomain is just the sum of the weights of the vertices in the subdomain, $|S_p| = \sum_{v \in S_p} |v|$. We denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c (note that $|E_c| = |L|$). Vertices which have an edge in E_c (i.e. those which are adjacent to vertices in another subdomain) are referred to as *border* vertices. 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 is assigned a subdomain S_p.

The definition of the graph partitioning problem is to find a partition which evenly balances the load or vertex weight in each subdomain whilst minimising the communications cost. To evenly balance the load, the optimal subdomain weight is given by $\overline{S} := \lceil |V|/P \rceil$ [1] and the *imbalance* 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). It is normal practice in graph partitioning to approximate the communications cost by $|E_c|$, the weight of cut edges or *cut-weight* and the usual (although not universal) definition of the graph partitioning problem is therefore to find π such that $S_p \leqslant \overline{S}$ and such that $|E_c|$ is minimised. Note that perfect balance is not always possible for graphs with non-unitary vertex weights.

2.2. The multilevel paradigm

In recent years it has been recognised that an effective way of both speeding up mesh partitioning techniques and/or, perhaps more importantly, giving them a global perspective is to use multilevel techniques.

The idea is to match pairs of vertices to form clusters, use the clusters 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 optimised on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated expansion/optimisation loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan–Lin (KL) [17], and other optimisation algorithms. The multilevel idea was first proposed by Barnard and Simon [2], as a method of speeding up spectral bisection and improved by both Hendrickson and Leland [11] and Bui and Jones [4], who generalised it to encompass local refinement algorithms. Several algorithms for carrying out the matching have been devised by Karypis and Kumar [15], while Walshaw and Cross [29] describe a method for utilising imbalance in the coarsest graphs to enhance the final partition quality.

2.2.1. Graph contraction

To create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ we use a variant of the edge contraction algorithm proposed by Hendrickson and Leland [11]. 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

[1] Where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x.

independence criterion. Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$.

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, matching each unmatched vertex with an unmatched neighbouring vertex (or with itself if no unmatched neighbours exist). Matched vertices are removed from the list. If there are several unmatched neighbours the choice of which to match with can be random, but it has been shown by Karypis and Kumar [15], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

2.2.2. The initial partition

Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel strategy is to carry out an initial partition. Here, following the idea of Gupta [10], we contract until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and then simply assign vertex i to subdomain S_i. Unlike Gupta, however, we do not carry out repeated expansion/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithm incorporates balancing, we commence on the expansion/optimisation sequence immediately.

2.2.3. Partition expansion

Having optimised the partition on a graph G_l, the partition must be interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter; if a vertex $v \in V_l$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_{l-1}$, will be in S_p.

2.3. The iterative optimisation algorithm

The iterative optimisation algorithm that we use at each graph level is a variant of the KL bisection optimisation algorithm which includes a hill-climbing mechanism to enable it to escape from local minima. Our implementation uses bucket sorting, the linear time complexity improvement of Fiduccia and Mattheyses [8], and is a partition optimisation formulation; in other words it optimises a partition of P subdomains rather than a bisection. It is fully described in [29].

The algorithm, as is typical for KL type algorithms, has inner and outer iterative loops with the outer loop terminating when no migration takes place during an inner loop. It uses two bucket sorting structures or bucket trees and is initialised by calculating the gain – the potential improvement in the cost function (in this context the cut-weight) – for all border vertices and inserting them into one of the bucket trees. These vertices are referred to as *candidate* vertices and the tree containing them as the *candidate tree*.

The inner loop proceeds by examining candidate vertices, highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration and then transferring it to the other bucket tree (the tree of *examined* vertices). If the candidate vertex is found acceptable, it is migrated, its neighbours have their gains updated and those which are not already in the examined tree are relocated in the candidate tree according to this updated gain. This inner loop terminates when the candidate tree is empty although it may terminate early if the partition cost rises too far above the cost of the best partition found so far. Once the inner loop has terminated, any vertices remaining in the candidate tree are transferred to the examined tree and finally pointers to the two trees are swapped ready for the next pass through the inner loop.

The algorithm also uses a KL type hill-climbing strategy; in other words vertex migration from subdomain to subdomain can be *accepted* even if it degrades the partition quality and later, based on the subsequent evolution of the partition, either rejected or *confirmed*. During each pass through the inner loop, a record of the optimal partition achieved by migration within that loop is maintained together with a list of vertices which have migrated since that value was attained. If subsequent migration finds a 'better' partition then the migration is *confirmed* and the list is reset. Note that it is possible to find better partitions despite selecting some vertices with negative gain because, as the optimiser runs, the gains of adjacent vertices will change and so the migration of a group of vertices some or all of which start with negative gain can in fact decrease the overall cost (i.e. produce a net positive gain). Once the inner loop is terminated, any vertices remaining in the list (vertices whose migration has not been confirmed) are migrated back to the subdomains they came from when the optimal cost was attained.

The algorithm, together with conditions for vertex migration acceptance and confirmation is fully described in [29].

2.4. Parallel multilevel graph partitioning

The parallel implementation of the multilevel graph partitioning strategy involves a number of fairly complex issues and coding difficulties [28]. However, the techniques are very similar in outline to the serial version and for the purposes of this paper, where the multilevel partitioner is used as a black box solver, the description above should give a sufficient overview of the multilevel paradigm. Both parallel and serial algorithms are implemented in a mesh partitioning tool known as JOSTLE which is freely available for academic and research purposes under a licensing agreement. [2]

3. Multiphase partitioning

In this section, we describe a strategy which addresses the multiphase partitioning problem, the principle of which is to partition each phase separately, phase by phase, but use the results of the previous phase to influence the partition of the current one. The partitioner which we use to carry out the partitioning of each phase is that described in Section 2 with a few minor modifications described in Section 3.3. However, in principle any partition optimisation algorithm could be used.

3.1. Vertex classification

To talk about multiphase partitioning and more specifically our methods for addressing the problem we need to first classify the graph vertices according to phase. For certain applications the mesh entities (e.g. nodes or elements) will each belong to one phase only (see for example Fig. 2(a) and also Section 4.1). However it is quite possible for a mesh entity, and hence the graph vertex representing it, to belong to more than one phase (see for example the application in Section 4.3, a contact–impact calculation where some mesh elements are involved in both contact and shell deformation phases). For this reason, if F is the number of phases, we require for each

[2] Available from `http://www.gre.ac.uk/jostle`

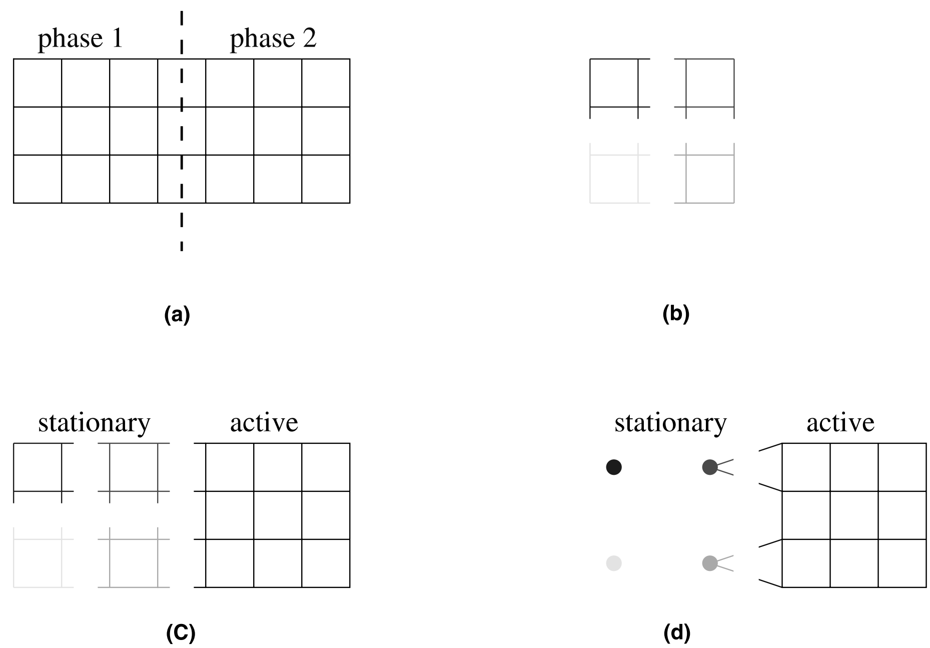

Fig. 2. Multiphase partitioning of a simple two phase mesh: (a) the two phases; (b) the partition of the type 1 vertices; (c) the input graph for the type 2 vertices; (d) the same input graph with stationary vertices condensed.

vertex v that the input graph includes a vector of length F, containing non-negative integer weights that represent the contribution of that vertex to the computational load in each phase. Thus if $|v|_i$ represents the contribution of vertex v to phase i then the weight vector for a vertex v is given by $\mathbf{w} = [|v|_1, |v|_2, \ldots, |v|_F]$ (this is exactly the same as for the multi-constraint paradigm of [16], see Section 3.6.3). For the example in Fig. 2(a) then, the phase 1 mesh nodes would be input with the vector $[1, 0]$ while the phase 2 nodes would be input with the vector $[0, 1]$ (assuming each node contributes a weight of 1 to their respective phases). We then define the vertex *type* to be the lowest value of i for which $|v|_i > 0$, i.e.

$$\text{type}(v) = \begin{cases} \min i & \text{such that } |v|_i > 0 \quad \text{for } i = 1, \ldots, F, \\ 0 & \text{if } |v|_i = 0 \qquad\qquad \text{for } i = 1, \ldots, F. \end{cases} \tag{1}$$

Thus in the case when the mesh phases are distinct (e.g. Fig. 2) the vertex type is simply the phase of the mesh entity that it represents; when the mesh entities belong to more than one phase then the vertex type is the first phase in which its mesh entity is active. Note that it is entirely possible that $|v|_i = 0$ for all $i = 1, \ldots, F$, (although this might appear to be unlikely it did in fact occur in the very first tests of the technique that we tried with a real application, see Section 4.3) and we refer to such vertices as type zero vertices. For clarification then, a mesh entity can belong to multiple phases, but the graph vertex which represents it can only be of one type $t = 0, \ldots, F$, where F is the number of phases.

3.2. Multiphase partitioning strategy

To explain the multiphase partitioning strategy, consider the example mesh shown in Fig. 2(a) which has two phases and which we require to partition into four subdomains. The basis of the strategy is to first partition the type 1 vertices, shown partitioned in Fig. 2(b) and then partition the type 2 vertices. However, we do not simply partition the type 2 vertices independent of the type 1 partition; to enhance data locality it makes sense to include the partitioned type 1 vertices in the calculation and use the graph shown in Fig. 2(c) as input for the type 2 partitioning. We retain the type 1 partition by requiring that the partitioner may not change the processor assignment of any type 1 vertex. We thus refer to those vertices which are not allowed to migrate (i.e. those which have already been partitioned in a previous phase) as *stationary* vertices. Vertices which belong to the current phase (non stationary ones) are referred to as *active*.

3.2.1. Vertex condensation

Because a large proportion of the vertices may be 'stationary' (i.e. the partitioner is not allowed to migrate them) it is rather inefficient to include all such vertices in the calculation. For this reason we condense all stationary vertices assigned to a processor p down to a single stationary *super-vertex* as shown in Fig. 2(d). This can considerably reduce the size of the input graph.

3.2.2. Graph edges

Edges between stationary and active vertices are retained to enhance the interphase data locality, however, as can be seen in Fig. 2(d), edges between the condensed stationary vertices are left out of the input graph. There is a good reason for this; our partitioner includes an integral load-balancing algorithm (to remove imbalance arising either from an existing partition of the input graph or internally as part of the multilevel process) which schedules load to be migrated along the edges of the subdomain graph. If the edges between stationary vertices are left in the input graph, then corresponding edges appear in the subdomain graph and hence the load-balancer may schedule load to migrate between these subdomains. However, if these inter subdomain edges arise *solely* because of the edges between stationary vertices then there may be no active vertices to realise this scheduled migration and the balancing may fail.

3.2.3. Summary

Although we have illustrated the multiphase partitioning algorithm with a two phase example, the technique can clearly be extended to arbitrary numbers of phases. Fig. 3 shows a pseudo-code description of the algorithm. Here the `partition phase graph` line is a call to the multilevel

```
for phase = 1,..,nphases {
        for each vertex v {
                if type(v) = phase {
                        include vertex in phase graph
                                with weight |v|_i
                } else if type(v) < phase {
                        p = partition(v)
                        condense v into stationary vertex p
                                adding weight |v|_i
                } else {
                        ignore v
                }
        }
        partition phase graph
}
```

Fig. 3. The multiphase partitioning algorithm.

single-phase partitioner and hence we can see that the multiphase mesh partitioning paradigm consists of a wrapper around a black box mesh partitioner. As the wrapper simply constructs a series of F subgraphs, one for each phase, implementation is straightforward.

3.3. Modifications to the multilevel partitioner

The modifications we need to make to the multilevel partitioner are relatively minor and simple to implement. Consider first of all the optimisation; all that we require is that the condensed stationary vertices do not migrate and we simply restrict them from doing so by not including them in the bucket sorting so that they cannot be considered for migration. This in turn leads to the modification for the graph contraction; since we do not allow stationary vertices to migrate, a cluster consisting of an active vertex matched with a stationary vertex will be prevented from migrating. Therefore, since we wish the active vertices to have total freedom to migrate, we do not allow them to match with stationary vertices. Furthermore stationary vertices are not allowed to match with each other since this would result in a cluster containing vertices in different subdomains. These modifications are easily achieved within the graph partitioner at each graph level by just matching each stationary vertex with itself – as if it had no unmatched neighbours – before any other matching takes place. Finally for the initial partition, the result of the matching graph contraction means that the coarsest graph consists of P stationary vertices each assigned to one processor and P (or more) unassigned active vertices. At this point we assign the active vertices with a simple greedy approach which takes account of gain, the aim being that if an active vertex is adjacent to one or more stationary vertices it should be assigned such that the cut-weight is minimised (i.e. assigned to the same processor as the stationary vertex with which it shares the heaviest edge).

3.4. Parallel issues

The parallel implementation of these techniques is relatively straight forward, if complex to code; once again, the wrapper around the black box multilevel partitioner is simply required to classify the vertices and construct a subgraph for each phase and this can be done in parallel. One major difference from the serial version, however, is that to execute in parallel the multiphase partitioner will already have a partitioned graph (because each processor will own a subset of the graph vertices) and the initial distribution can be very crude, e.g. [28], and bear no relation to the multiple computational phases. Indeed, for a dynamic version of the technique (not tested here), where the partitioner may be required to reuse an existing partition in order to minimise data migration, this issue arises even for the serial multiphase partitioner and results in extra requirements discussed in the next section.

3.5. Extensions to the multilevel partitioner

In order to function correctly on multiple phase based subgraphs the black box multilevel partitioner does require some additional functionality. In particular the partitioner needs to be able to correctly handle disconnected graphs (and, as a special case, isolated vertices) and it requires a mechanism for seeding empty subdomains (for example if one of the existing subdomains initially has no type 1 vertices). In fact, all of the required functionality has been part of the JOSTLE partitioning tool for some time; we omit the implementation details here for brevity but a full description can be found in [31].

3.6. Related work

3.6.1. Graph manipulation approaches

Although, as far as we are aware, the techniques described in this paper are novel, they do resemble in certain respects approaches used to address some other mesh/graph partitioning problems. In particular, the strategy of modifying the graph and then using standard partitioning techniques has been successfully employed previously. For example, Walshaw and Berzins [32] condensed graph vertices together to form 'super-vertices', one per subdomain and then employed the standard recursive spectral bisection algorithm, [27], in order to prevent excessive data migration for dynamic repartitioning of adaptive meshes. In a similar vein, both Hendrickson and Leland [13], and Pellegrini and Roman [25], used additional graph vertices, essentially representing processors/subdomains in order to enhance data locality when mapping onto parallel machines with non-uniform interconnection architectures (e.g. a grid of processors or a metacomputer). The multiphase partitioning strategy is another in this broad class of graph manipulation approaches.

3.6.2. Contact–impact simulations

One of the particular areas of interest driving the development of multiphase partitioning algorithms has been the use of contact–impact algorithms (for example in the automotive industry for simulating crashes, e.g. [5]). Typically the simulation will involve localised stress–strain finite element calculations over the entire mesh together with a much more complex contact–impact detection phase over the restricted areas of possible penetration, [20]. It is usually the imbalance introduced during this contact phase which is responsible for serious deterioration in the overall scalability of the code and several approaches to overcome it have been tried.

Vertex weighting. One strategy arising from crashworthiness simulations and designed by Clinckemaillie, Lonsdale et al. [5,20], to address this problem, is to use a static partition of the mesh but to add contact-related weights to the partitioning cost function for vertices that are part of a contact surface. Although this approach does not directly address the two-phase nature of the problem, a significant improvement was reported over the version which simply partitioned the stress–strain mesh.

Overpartitioning. Another technique arising from crashworthiness simulations and again involving a static partition of the mesh, is that employed by Galbas and Kolp [9], and known as overpartitioning. In this approach the mesh is split into many more subdomains than there are processors (typically 4 or 8 times as many) with the aim that parts of the mesh that will be involved in contact are split into sufficient numbers of subdomains to achieve balance. Dynamic load-balance is attained by (re)assigning subdomains to processors such that each processor has an equal share of workload from each phase. A disadvantage is that the communications cost will rise due to the increased number of interface mesh nodes, but the authors report performance improvements of up to 50% over a version of the code which does not use overpartitioning.

Multiple partitions. A third strategy, proposed by Hendrickson et al. [14,26], is to use two different partitions of the mesh, one for the contact detection and the other for the finite element calculation. Indeed the contact detection part uses the rapid recursive coordinate bisection algorithm in a *dynamic* sense (in that the partition is generated anew each time-step). A disadvantage of this approach is that information must be communicated between the two partitions at every time-step and that some memory is duplicated, however the authors report that the advantage of achieving load-balance in both phases greatly outweighs the cost of maintaining two partitions.

3.6.3. The multi-constraint partitioning problem

Most closely related to the work presented here is the multi-constraint partitioning method of Karypis and Kumar [16], a different and in some ways more general approach that can be applied to the multiphase partitioning problem. The idea is to view the problem as a graph partitioning problem with multiple constraints (in this case load-balancing constraints). Once again the vertices of the graph have a vector of weights, in this case representing the contribution to each balancing constraint. However, in contrast to the methods presented here, Karypis and Kumar solve the problem in a single multilevel computation (rather than on a phase by phase basis). The multilevel methods are then modified in a number of ways. Firstly, during the contraction procedure the matching is driven by trying to create vertex clusters with balanced weights in each phase. Thus a vertex with weight vector $[1, 0]$ would match with an adjacent vertex with weight vector $[0, 1]$ in preference to vertices with weights $[1, 0]$ or $[1, 1]$ in order to create a balanced cluster. The refinement phase meanwhile uses greedy refinement which migrates vertices between subdomains if the movement improves the partition quality subject to the balancing constraints or improves the balance without worsening the quality (in this context the cut-weight). The initial partitioning is done with recursive multilevel bisection (once the coarsened graph is smaller than $50P$ vertices) and uses multiple queues to satisfy the constraints.

The results presented in [16] suggest that this approach is well able to handle the multiple constraints and provides partition qualities around 20–70% worse than a single constraint algorithm (acting on the same graph without multiple weights) and the partition takes around 1.5–3 times longer to compute. These are not unreasonable overheads given the additional complexity of the problem. However, the problems on which Karypis and Kumar test their algorithms are somewhat artificial and so is it difficult to draw any meaningful conclusions (for example they do not test a simple two-phase problem, e.g. see Section 4.1).

This multi-constraint paradigm is a more general approach than the multiphase strategy presented here since it could, for example, be applied to the problem of trying to balance computational and memory requirements where, say, the weight vector for vertex v, $[|v|_1, |v|_2]$ has entries $|v|_1$ which represents the computational cost and $|v|_2$ which represents the memory requirement. Our approach, on the other hand, requires that $|v|_i = 0$ for at least some $v \in V$ and $i = 1, \dots, F$. However the results in Section 4 suggest that our more focussed approach can work well on the (large) subset of multiphase problems for which it is designed. Also the multiphase approach is somewhat simpler to implement since it merely involves a wrapper around the multilevel partitioner and can, in principle, reuse existing software features and components such as, for example, dynamic load-balancing techniques, [30].

3.6.4. Separator theory for graphs with multiple weights

A separator for a graph is a small set of vertices or edges whose removal divides the graph into disjoint pieces of approximately equal size. A class of graphs is said to satisfy an $f(N)$ vertex separator theorem if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph of N vertices in the class has a separator of at most $\beta f(N)$ vertices whose removal leaves no connected component with more than αN vertices. Several useful classes of graphs can be shown to satisfy separator theorems; for example, Lipton and Tarjan showed, [18], that planar graphs have an $O(N^{1/2})$ vertex separator which partitions the graph into two sets whose size is at least $N/3$. In [6,7], Djidjev and Gilbert extended the result to show that graphs which satisfy an N^{λ} vertex separator theorem also satisfy the same theorem if multiple weights are attached to each vertex. Karypis and Kumar have also investigated the same issue, [16], although not achieving such a strong result.

In the context of this paper we are interested in edge separators (the cut-edges can be referred to as an edge separator). Of course it is possible to find an edge separator for a graph $G(V, E)$ by finding a vertex separator of the dual graph $G'(E, E')$ where each edge $e \in E$ of G is represented by a vertex in G' and there are dual edges $e' \in E'$ for every pair of edges $e_1, e_2 \in E$ incident on the same vertex $v \in V$ (i.e. $e_1 = (v, u_1)$ and $e_2 = (v, u_2)$ for some vertices $u_1, u_2 \in V$ with $u_1 \neq u_2$). However, such dual graphs, even if derived from simple 2D planar FE and FV meshes, are not themselves planar and so the separator theory, although of interest, is of limited use here.

4. Experimental results

In this section, we test the multiphase partitioning strategy on three different sorts of multiphase mesh partitioning problems (MMPPs). We do not test the algorithms exhaustively; it is not too difficult to derive MMPPs, pathological and otherwise, for which the multiphase partitioning strategy will fail (e.g. for the same reasons that the multiconstraint paradigm is more general, see above Section 3.6.3). However, we do attempt to demonstrate that there is a fairly large class of problems for which standard mesh partitioning techniques will completely fail to balance individual computational phases, but for which the multiphase approach can achieve high quality partitions.

4.1. Distinct phase results

The first set of experiments are performed on a set of artificial but not unrealistic examples of distinct two-phase problems. By distinct we mean that the computational phase regions do not overlap and are separated by a relatively small interface. Such problems are typical of many multiphysics computational mechanics applications such as solidification, e.g. [1].

The problems are constructed by taking a set of 2D and 3D meshes, some regular grids and some with irregular (or unstructured) adjacencies and geometrically bisecting them so that one half is assigned to phase 1 and the other half to phase 2. Table 1 gives a summary of the mesh sizes and classification, where V_1 and V_2 represent the number of type 1 and type 2 vertices, respectively, and E is the number of edges. These are possibly the simplest form of two-phase problems and provide a clear demonstration of the need for multiphase mesh partitioning.

We have tested the meshes with three different partitioners for three different values of P, the number of sub-domains/processors. The first of these partitioners, JOSTLE-S, is simply the standard multilevel mesh partitioner JOSTLE, [29], which takes no account of the different phases. The multiphase version of jostle, JOSTLE-M and the parallel multiphase version, PJOSTLE-M, incorporate the multiphase partitioning paradigm as described in this paper.

The results in Table 2 show for each mesh and value of P the proportion of cut edges, $|E_c|/|E|$, (which gives an indication of the partition quality in terms of communication overhead) and the imbalance for the two phases, λ_1 and λ_2, respectively. These three quality metrics are then averaged for each partitioner and value of P.

Table 1
Distinct phase meshes

Name	V_1	V_2	E	Description
512 × 256	65 536	65 536	261 376	2D Regular grid
crack	4195	6045	30 380	2D Nodal mesh
dime20	114 832	110 011	336 024	2D Dual mesh
64 × 32 × 32	32 768	32 768	191 488	3D Regular grid
brack2	33 079	29 556	366 559	3D Nodal mesh
mesh100	51 549	51 532	200 976	3D Dual mesh

Table 2
Distinct phase results

Mesh	$P = 4$			$P = 8$			$P = 16$		
	$\|E_c\|/\|E\|$	λ_1	λ_2	$\|E_c\|/\|E\|$	λ_1	λ_2	$\|E_c\|/\|E\|$	λ_1	λ_2
	JOSTLE-S: jostle single-phase								
512 × 256	0.004	2.000	2.000	0.006	2.000	2.000	0.011	2.000	2.000
crack	0.015	1.906	1.614	0.026	2.434	1.692	0.041	2.445	1.709
dime20	0.001	1.881	1.726	0.003	1.986	2.036	0.004	1.972	2.049
64 × 32 × 32	0.023	2.000	2.000	0.038	2.000	2.000	0.052	2.000	2.000
brack2	0.008	1.932	2.096	0.023	1.937	2.138	0.037	1.949	2.145
mesh100	0.008	2.012	1.987	0.016	2.011	2.015	0.025	2.034	2.005
Average	0.010	1.955	1.904	0.019	2.061	1.980	0.028	2.067	1.985
	JOSTLE-M: jostle multiphase								
512 × 256	0.004	1.025	1.026	0.009	1.028	1.019	0.013	1.028	1.026
crack	0.016	1.025	1.027	0.030	1.025	1.028	0.055	1.027	1.029
dime20	0.002	1.027	1.015	0.003	1.020	1.025	0.006	1.016	1.018
64 × 32 × 32	0.027	1.026	1.029	0.041	1.030	1.029	0.063	1.026	1.030
brack2	0.021	1.010	1.014	0.034	1.030	1.030	0.052	1.029	1.026
mesh100	0.011	1.023	1.021	0.020	1.022	1.029	0.034	1.023	1.029
Average	0.013	1.023	1.022	0.023	1.026	1.027	0.037	1.025	1.026
	PJOSTLE-M: parallel jostle multiphase								
512 × 256	0.006	1.000	1.000	0.010	1.000	1.000	0.016	1.000	1.001
crack	0.016	1.000	1.000	0.036	1.000	1.001	0.055	1.000	1.000
dime20	0.002	1.000	1.000	0.004	1.000	1.000	0.007	1.001	1.001
64 × 32 × 32	0.029	1.000	1.000	0.046	1.000	1.002	0.066	1.002	1.013
brack2	0.020	1.000	1.001	0.033	1.000	1.002	0.052	1.001	1.005
mesh100	0.011	1.000	1.000	0.021	1.000	1.000	0.033	1.002	1.001
Average	0.014	1.000	1.000	0.025	1.000	1.001	0.038	1.001	1.004

As suggested, JOSTLE-S, whilst achieving the best minimisation of cut-weight, completely fails to balance the two phases (since it takes no account of them). On average (and as one might expect from the construction of the problem) the imbalance is approximately two, i.e. the largest subdomain is twice the size that it should be and so the application might be expected to run twice as slowly as a well partitioned version (neglecting any communication overhead). This is because the single phase partitioner ignores the different graph regions and (approximately) partitions each phase between half of the processors. Both the multiphase partitioners, however, manage to achieve good balance, although note that all the partitioners have an imbalance tolerance, set at run-time, of 1.03, i.e. any imbalance below this is considered negligible. This is particularly noticeable for the serial version, JOSTLE-M, which, because of its global nature is able to utilise the imbalance tolerance to achieve higher partition quality (see [29]) and thus results in imbalances close to (but not exceeding) the threshold of 1.03. The parallel partitioner, PJOSTLE-M, on the other hand, produces imbalances much closer to 1.0 (perfect balance).

In terms of the cut-weight, JOSTLE-M produces partitions about 28% worse on average than JOSTLE-S and those of PJOSTLE-M are about 35% worse. These are to be expected as a result of the more complex partitioning problem and are in line with the 20–70% deterioration reported by Karypis and Kumar [16] for their multi-constraint algorithm.

We do not show run time results here and indeed the multiphase algorithm is not particularly time-optimised but, for example, for ‘mesh100’ and $P = 16$, the run times on a DEC Alpha workstation were 3.30 s for JOSTLE-M and 2.22 s for JOSTLE-S. For the same mesh in parallel on a Cray T3E (with slower processors) the run times were 5.65 s for PJOSTLE-M and 3.27 for

PJOSTLE-S (the standard single-phase parallel version described in [28]). On average the JOSTLE-M results were about 1.5 times slower than those of JOSTLE-S and PJOSTLE-M was about two times slower than PJOSTLE-S. This is well in line with the 1.5–3 times performance degradation suggested for the multi-constraint algorithm, [16].

4.2. Multiple mesh entities

The second set of test examples arise again from two phase problems but in this set of experiments the phases are not well separated with a small interface as above, but highly integrated and very interconnected. This type of multiphase problem can easily arise for a solver in which different calculations take place on mesh nodes from those taking place on mesh elements and the two calculations are separated by global synchronisation points in the solver. This issue is discussed in [24] and we simulate it taking a set of meshes and assigning the elements to phase 1 and the nodes to phase 2 (although similar results, not shown here, are achieved if the assignment is reversed).

The set of four meshes are summarised in Table 3 with V_1 representing the number of mesh elements and V_2 the number of mesh nodes. Again E is the number of edges.

Table 4 shows the partitioning results in the same form as Table 2. Interestingly, the single phase algorithm, JOSTLE-S, actually does a very good job for the 2D meshes, balancing both

Table 3
Node/element meshes

Name	V_1	V_2	E	Description
4elt	30 269	15 606	181 614	2D Triangular mesh
t60k	60 005	30 570	360 030	2D Triangular mesh
cs4	22 499	4083	161 574	3D Tetrahedral mesh
mesh100	103 081	20 596	742 162	3D Tetrahedral mesh

Table 4
Node/element results

Mesh	$P=4$			$P=8$			$P=16$		
	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2
	JOSTLE-S: jostle single-phase								
4elt	0.008	1.000	1.001	0.010	1.003	1.007	0.016	1.012	1.018
t60k	0.003	1.001	1.003	0.008	1.002	1.004	0.014	1.005	1.014
mesh100	0.015	1.030	1.036	0.029	1.019	1.042	0.044	1.013	1.079
cs4	0.051	1.008	1.076	0.073	1.021	1.084	0.105	1.018	1.086
Average	0.019	1.010	1.029	0.030	1.011	1.034	0.045	1.012	1.049
	JOSTLE-S: jostle multiphase								
4elt	0.007	1.018	1.016	0.010	1.019	1.028	0.017	1.021	1.025
t60k	0.003	1.004	1.003	0.008	1.014	1.019	0.015	1.020	1.024
mesh100	0.015	1.029	1.029	0.029	1.028	1.030	0.049	1.026	1.029
cs4	0.057	1.028	1.029	0.077	1.028	1.027	0.119	1.016	1.023
Average	0.021	1.020	1.019	0.031	1.022	1.026	0.050	1.021	1.025
	PJOSTLE-M: parallel jostle multiphase								
4elt	0.006	1.000	1.000	0.011	1.000	1.000	0.019	1.001	1.005
t60k	0.004	1.000	1.000	0.008	1.000	1.000	0.016	1.000	1.003
mesh100	0.019	1.000	1.000	0.034	1.000	1.009	0.054	1.000	1.014
cs4	0.054	1.000	1.008	0.081	1.000	1.016	0.114	1.000	1.023
Average	0.021	1.000	1.002	0.033	1.000	1.006	0.051	1.000	1.011

mesh elements and nodes well. This is not too surprising since the type 1 and type 2 graph vertices (the mesh elements and nodes) are closely integrated and any reasonably compact subdomain is likely to contain an equal share of both. However for the 3D meshes, with their more complex distribution patterns and relatively much smaller proportion of nodes to elements, this coincidence starts to break down and although the elements are well balanced, the mesh nodes are not that well balanced (e.g. 8.6% imbalance for mesh 'cs4', $P = 16$), confirming the issues raised in [24].

The multiphase results again bear out the trends seen in Table 2; the multiphase partitioners balance both phases well with the parallel version, PJOSTLE-M achieving the best balances. Meanwhile the cut-weight is even closer to that attained by the single-phase algorithm and, respectively, the results of JOSTLE-M and PJOSTLE-M are just 8.5% and 11.7% worse than JOSTLE-S. This relative closeness is a function of the fairly even distribution of the nodes and elements throughout the mesh.

Again, we do not show run time results here but, for example, for 't60k' and $P = 16$, the run times on a DEC Alpha workstation were 2.65 s for JOSTLE-M and 1.88 s for JOSTLE-S. For the same mesh in parallel on a Cray T3E (with slower processors) the run times were 4.39 s for PJOSTLE-M and 3.62 for PJOSTLE-S. On average the JOSTLE-M results were about 1.25 times slower than those of JOSTLE-S and PJOSTLE-M was about 1.1 times slower than PJOSTLE-S. This is better than the 1.5–3 times performance degradation suggested for the multi- constraint algorithm, [16], although Karypis and Kumar did not test this type of problem there.

4.3. Contact–impact results

The third set of test meshes arise from an industrial application and are some examples of contact–impact simulations. This sort of problem has been discussed in Section 3.6.2 and the load-balancing issues and cost modelling have been investigated in detail by the DRAMA project, [3,19,21].

The test meshes are summarised in Table 5, where V_1 represents the number of graph vertices involved in the contact phase and V_2 the number involved in the stress–strain finite element calculations (again see [3,19,21] for further details of how these graphs are constructed and weighted). They were generated by the PAM-CRASH code, [5], shortly after contact had occurred. As a result the areas of contact consist of many scattered penetration nodes (mesh nodes where two different parts of the mesh interpenetrate) as the metal shell under simulation starts to buckle. Thus, the type 1 vertices are distributed between many disconnected regions (up to 244 regions in the case of the bmw mesh). This results in an extremely complex partitioning problem.

The partitioning results are shown in Table 6 and it can be seen that, with one exception (PJOSTLE-M for the 'box' mesh, $P = 16$), the multiphase partitioners achieve load-balance within the tolerance of 1.03.

We are not entirely sure why the exception occurs with the box mesh but believe it to be as a result of the size and nature of the single phase balancing problem for the type 1 vertices rather than anything inherent in the multiphase strategy. In fact this unrealistically small example (just

Table 5
Contact–impact meshes

Name	V_1	V_2	E	Description
box	488	3882	9242	3D Box beam crumpling simulation
audi	2750	53 071	112 597	3D AUDI car crash simulation
bmw	5508	95 534	208 157	3D BMW car crash simulation

Table 6
Contact–impact results

Mesh	$P = 4$			$P = 8$			$P = 16$		
	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2	$\lvert E_c\rvert/\lvert E\rvert$	λ_1	λ_2
	JOSTLE-S: jostle single-phase								
box	0.026	2.038	1.047	0.039	3.985	1.082	0.059	4.175	1.117
audi	0.003	2.092	1.046	0.009	3.777	1.050	0.012	6.129	1.075
bmw	0.005	2.858	1.048	0.009	5.151	1.044	0.014	5.774	1.048
Average	0.011	2.329	1.047	0.019	4.304	1.059	0.028	5.359	1.080
	JOSTLE-S: jostle multiphase								
box	0.027	1.029	1.010	0.064	1.029	1.027	0.115	1.021	1.023
audi	0.012	1.027	1.027	0.017	1.022	1.029	0.028	1.025	1.029
bmw	0.010	1.019	1.026	0.019	1.029	1.030	0.027	1.029	1.030
Average	0.016	1.025	1.021	0.033	1.027	1.029	0.057	1.025	1.027
	PJOSTLE-M: parallel jostle multiphase								
box	0.028	1.013	1.000	0.070	1.036	1.012	0.107	1.265	1.016
audi	0.011	1.011	1.001	0.017	1.024	1.002	0.026	1.028	1.010
bmw	0.010	1.002	1.000	0.021	1.010	1.005	0.038	1.019	1.031
Average	0.016	1.009	1.000	0.036	1.023	1.006	0.057	1.104	1.019

488 vertices distributed into 16 subdomains) is difficult enough (though possible) for a parallel partitioner in view of the fine granularity. However it is significantly complicated by the fact that the type 1 vertices are weighted with values ranging from 0 to 842 013 (these weights arising from the cost modelling within DRAMA, [3]). In particular the zero weight vertices pose a challenge for a diffusive style load-balancer (such as lies at the heart of JOSTLE) as no gain is accrued by moving them and thus the balancing is 'directionless'. Fortunately, however, such problems are not encountered on the larger (and more realistic) meshes 'audi' and 'bmw'.

The single-phase version, JOSTLE-S, completely fails to achieve balance, particularly with the contact nodes, which, although are scattered mainly occur in the front portion of each mesh (where the impact has taken place). However the shell nodes are not well balanced either.

In terms of cut weight, JOSTLE-M and PJOSTLE-M achieve results which are about two times worse than JOSTLE-S (82% and 87% worse respectively). Again, this reflects the highly complex nature of the partitioning problem.

Again, we do not show run time results here but, for example, for the audi mesh and $P = 16$, the run times on a DEC Alpha workstation were 1.80 s for JOSTLE-M and 1.02 s for JOSTLE-S. For the same mesh in parallel on a Cray T3E (with slower processors) the run times were 2.98 s for PJOSTLE-M and 2.33 for PJOSTLE-S. On average the JOSTLE-M results were about 1.9 times slower than those of JOSTLE-S and PJOSTLE-M was about 1.6 times slower than PJOSTLE-S. This again compares well with the 1.5–3 times performance degradation suggested for the multi-constraint algorithm, [16].

5. Summary and future research

We have described a new approach for addressing the load-balancing issues of CM codes containing multiple computational phases. This approach, the multiphase mesh partitioning strategy, consists of a graph manipulation wrapper around an almost unmodified black box multilevel mesh partitioner, JOSTLE, which is used to partition each phase individually. As such the strategy is relatively simple to implement and could, in principle, reuse existing features of the partitioner, such as minimising data migration in dynamic repartitioning context.

We have tested the strategy on examples of MMPPs arising from three different applications and demonstrated that it can succeed in producing high quality, *balanced* partitions where a standard mesh partitioner simply fails (as it takes no account of the different phases). The multiphase partitioner does take somewhat longer than the single phase version, typically 1.5–2 times as long. This corresponds to the general optimisation rule of thumb that harder problems take longer to optimise (e.g. see [28]) however we do not believe that this relationship can be quantified in any meaningful way. We have not tested the strategy exhaustively and acknowledge that it is not too difficult to derive MMPPs for which it will not succeed. In fact, in this respect it is like many other heuristics (including most mesh partitioners) which work for a broad class of problems but for which counter examples to any conclusions can often be found.

Some examples of the multiphase mesh partitioning strategy in action for contact–impact problems can be found in [3], but with regard to future work in this area, it would be useful to investigate its performance in a variety of other genuine CM codes. In particular, it would be useful to look at examples for which it does not work and either try and address the problems or at least characterise what features it cannot cope with.

More specifically we are particularly interested in looking at better ways of joining disconnected regions and we believe that this would enhance the performance of the strategy for the contact–impact problems (Sections 3.6.2 and 4.3). Currently this is achieved with a somewhat random approach and we believe that this could be improved by incorporating geometric information.

References

[1] C. Bailey, P. Chow, M. Cross, Y. Fryer, K. Pericleous, Multiphysics modelling of the metals casting process, Proc. R. Soc. Lond. A 452 (1995) 459–486.

[2] S.T. Barnard, H.D. Simon, A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, Concurrency: Practice & Experience 6 (2) (1994) 101–117.

[3] A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, C. Walshaw, Dynamic multi-partitioning for parallel finite element applications, in: E.H. D'Hollander, et al. (Eds.), Proc. Parallel Computing: Fundamentals and Applications, Proc. Intl. Conf. Parco '99, Delf, The Netherlands (Imperial College Press, 2000) pp. 259–266.

[4] T.N. Bui, C. Jones. A heuristic for reducing fill-in in sparse matrix factorization, in: R.F. Sincovec, et al. (Eds.), Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1993, pp. 445–452.

[5] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, M. Holzner, Performance issues of the parallel PAM-CRASH code, Int. J. Supercomputer Appl. High Performance Comput. 11 (1) (1997) 3–11.

[6] H. Djidjev, J. Gilbert, Separators in graphs with negative and multiple vertex weights, TR 94-226, Department of Computer Science, Rice University, 1994.

[7] H. Djidjev, J. Gilbert, Separators in graphs with negative and multiple vertex weights, Algorithmica 23 (1) (1999) 57–71.

[8] C.M. Fiduccia, R.M. Mattheyses, A linear time heuristic for improving network partitions, in: Proceedings of the 19th IEEE Design Automation Conference, IEEE, Piscataway, NJ, 1982, pp. 175–181.

[9] H.G. Galbas, O. Kolp, Dynamic load balancing in crashworthiness simulation, in: J. Palma, et al. (Eds.), Vector and Parallel Processing – VECPAR '98, LNCS, vol. 1573, Springer, Berlin, 1999, pp. 263–270 (selected paper from Third International Conference, Porto, Portugal).

[10] A. Gupta, Fast and effective algorithms for graph partitioning and sparse matrix reordering, IBM J. Res. Development 41 (1/2) (1996) 171–183.

[11] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, Technical Report SAND 93-1301, Sandia National Labs, Albuquerque, NM, 1993.

[12] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: S. Karin (Ed.), Proceedings of the Supercomputing '95, San Diego, ACM Press, New York, 1995.

[13] B. Hendrickson, R. Leland, R. Van Driessche, Enhancing data locality by using terminal propagation, in: Proceedings of the 29th Hawaii International Conference System Science, 1996.
[14] B. Hendrickson, S. Plimpton, S. Attaway, C. Vaughan, D. Gardner, A new parallel algorithm for contact detection in finite element methods, in: M. Heath, et al. (Eds.), Proceedings of the High Performance Computing '96. SIAM, Philadelphia, PA, 1996.
[15] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[16] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint graph partitioning, TR 98-019, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, 1998.
[17] B.W. Kernighan, S. Lin, An efficient heuristic for partitioning graphs, Bell Syst. Tech. J. 49 (1970) 291–308.
[18] R.J. Lipton, R.E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36 (1979) 177–189.
[19] G. Lonsdale, A. Basermann, J. Fingberg, T. Coupez, H. Digonnet, J. Clinckemaillie, G. Thierry, P. Dannaux, B. Maerten, D. Roose, R. Ducloux, C. Walshaw, DRAMA: dynamic re-allocation of meshes for parallel finite element applications, in: B.H.V. Topping (Ed.), Developments in Computational Mechanics with High Performance Computing, Civil-Comp Press, Edinburgh, 1999, pp. 61–66 (Proc. Parallel & Distributed Computing for Computational Mechanics, Weimar, Germany, 1999).
[20] G. Lonsdale, B. Elsner, J. Clinckemaillie, S. Vlachoutsis, F. de Bruyne, M. Holzner, Experiences with industrial crashworthiness simulation using the portable, message-passing PAM-CRASH code, in: High-Performance Computing and Networking (Proc. HPCN '95), LNCS, vol. 919, Springer, Berlin, 1995, pp. 856–862.
[21] B. Maerten, D. Roose, A. Basermann, J. Fingberg, G. Lonsdale, DRAMA: a library for parallel dynamic load balancing of finite element applications, in: Parallel Processing for Scientific Computing. SIAM, Philadelphia, PA, 1999 (CD-ROM).
[22] K. McManus, M. Cross, S. Johnson, Integrated flow and stress using an unstructured mesh on distributed memory parallel systems, in: N. Satofuka, et al. (Eds.), Parallel Computational Fluid Dynamics: New Algorithms and Applications, Elsevier, Amsterdam, 1995, pp. 287–294 (Pro. Parallel CFD '94, Kyoto, 1994).
[23] K. McManus, C. Walshaw, M. Cross, S. Johnson, Unstructured mesh computational mechanics on DM parallel platforms, Z. Angew. Math. Mech. 76 (S4) (1996) 109–112.
[24] K. McManus, C. Walshaw, S. Johnson, M. Cross, Partition alignment in three dimensional un-structured mesh multi-physics modelling, in: C.A. Lin, et al. (Eds.), Parallel Computational Fluid Dynamics: Development & Applications of Parallel Technology, Elsevier, Amsterdam, 1999, pp. 459–466 (Proc. Parallel CFD '98, Taiwan, 1998).
[25] F. Pellegrini, J. Roman, Experimental analysis of the dual recursive bipartitioning algorithm for static mapping, TR 1038-96, LaBRI, URA CNRS 1304, University of Bordeaux I, 351, cours de la Libération, 33405 TALENCE, France, 1996.
[26] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, D. Gardner, Transient dynamics simulations: parallel algorithms for contact detection and smoothed particle hydrodynamics, J. Parallel Distrib. Comput. 50 (1998) 104–122.
[27] H.D. Simon, Partitioning of unstructured problems for parallel processing, Comput. Syst. Engrg. 2 (1991) 135–148.
[28] C. Walshaw, M. Cross, Parallel optimisation algorithms for multilevel mesh partitioning, Parallel Comput. 26 (12) (2000) 1635–1660 (originally published as University of Greenwich, Technical Report 99/IM/44).
[29] C. Walshaw, M. Cross, Mesh partitioning: a multilevel balancing and refinement algorithm, SIAM J. Sci. Comput. 22 (1) (2000) 63–80 (originally published as University of Greenwich, Technical Report 98/IM/35).
[30] C. Walshaw, M. Cross, M. Everett, Parallel dynamic graph partitioning for adaptive unstructured meshes, J. Parallel Distrib. Comput. 47 (2) (1997) 102–108 (originally published as University of Greenwich, Technical Report 97/IM/20).
[31] C. Walshaw, M. Cross, K. McManus, Multiphase mesh partitioning. Technical Report 99/IM/51, University of Greenwich, London SE10 9LS, UK, November 1999.
[32] C.H. Walshaw, M. Berzins, Dynamic load-balancing for PDE solvers on adaptive unstructured meshes, Concurrency: Practice Experience 7 (1) (1995) 17–28 (originally published as University of Leeds, Technical Report 92.32, 1992).