Multiphase Mesh Partitioning for Parallel Computational Mechanics Codes

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

School of Computing and Mathematical Sciences, University of Greenwich,
Old Royal Naval College, Greenwich, London, SE10 9LS, UK.
C.Walshaw@gre.ac.uk, URL: http://www.gre.ac.uk/~c.walshaw

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 example meshes containing multiple computational phases and it is demonstrated that our method can achieve high quality partitions where a standard mesh partitioning approach fails.

Keywords: graph-partitioning, load-balancing, parallel multiphysics.

1 Introduction

The need for mesh partitioning arises naturally in many finite element and finite volume 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-hard (i.e. cannot be solved in polynomial time, [3]), 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. [4].

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

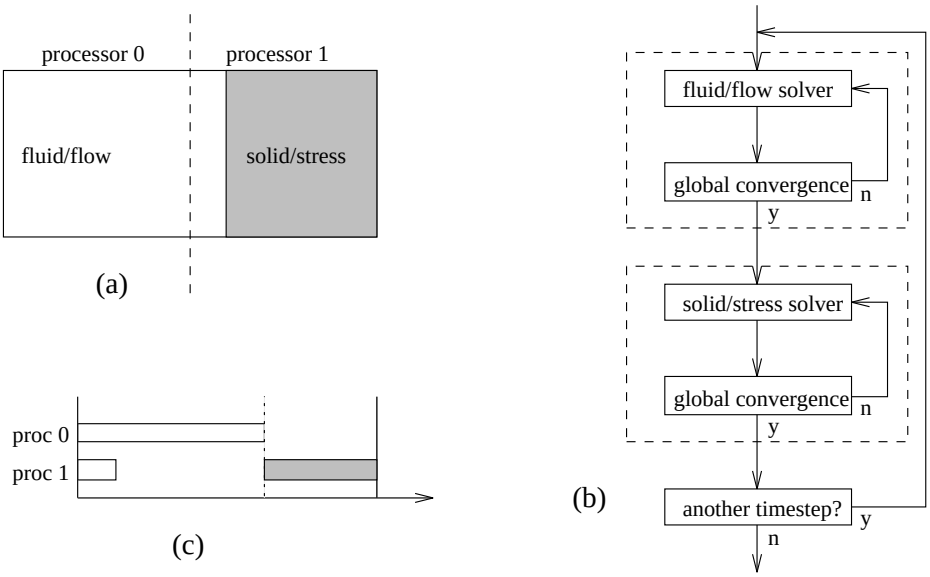

Fig. 1. An example of a multiphysics problem

or tetrahedra, or the mesh nodes). Even in the 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 but 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 Figure 1(a) with a partition for 2 processors indicated by the dotted line. This partition might normally be considered of good quality but for the solution algorithm in Figure 1(b) it is completely unsuitable. As Figure 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 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 parti-

tioning problem. If, for example, all the loops in Figure 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 Figure 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 & stress regions might need to be weighted differently. In the simple example in Figure 1 an obvious (and relatively good) load-balancing strategy, therefore, is simply to partition each region (i.e. liquid & solid) of the domain separately so that each processor has an equal number of entities from each region. However, in more complex cases, for example where the regions relating to different computational phases overlap, this may fail to provide a good solution and an advanced strategy is required.

We refer to this modified 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. [7]) 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. [6], 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 a strategy for dealing with MMPPs, which uses existing single-phase mesh partitioning algorithms as 'black box' solvers, to partition the problem phase by phase, each partition based on those of the previous phases. The details of this approach are described in Section 2, in particular the necessary vertex classification scheme (§2.1) and an outline of the implementation (§2.2). In Section 3 we present results for the techniques on an illustrative set of example MMPPs. Finally, in Section 4 we summarise the paper and mention some suggestions for further research.

Restrictions on space preclude a full discussion of related work (although some different approaches are reviewed in [10]). However most closely related to the work presented here is the multi-constraint partitioning method of Karypis & Kumar, [5], a different and in some ways more general approach that can be applied to the multiphase partitioning problem. Their idea is to view the problem as a graph partitioning problem with multiple constraints (in this case load-balancing constraints). As here 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 & Kumar solve the problem in a single computation (rather than on a phase by phase basis).

2 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, but use the results of previous phases to influence the partition of the current one. The partitioner which we use to carry out the partitioning of each phase is outlined in [10]; however, in principle any partition optimisation algorithm could be used.

2.1 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, e.g. Figure 2(a), but it is quite possible for a mesh entity, and hence the graph vertex representing it, to belong to more than one phase. For this reason, if F is the number of phases (i.e. the number of distinct computational subphases separated by global synchronisation points – see §1.1), we require for each 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 Karypis & Kumar, [5]). For the example in Figure 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.

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

Thus in the case when the mesh phases are distinct (e.g. Figure 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, [10]) 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.

2.2 Multiphase partitioning strategy

To explain the multiphase partitioning strategy, consider the example mesh shown in Figure 2(a) which has two phases and for which we are required to partition the mesh nodes into 4 subdomains. The basis of the strategy is to first

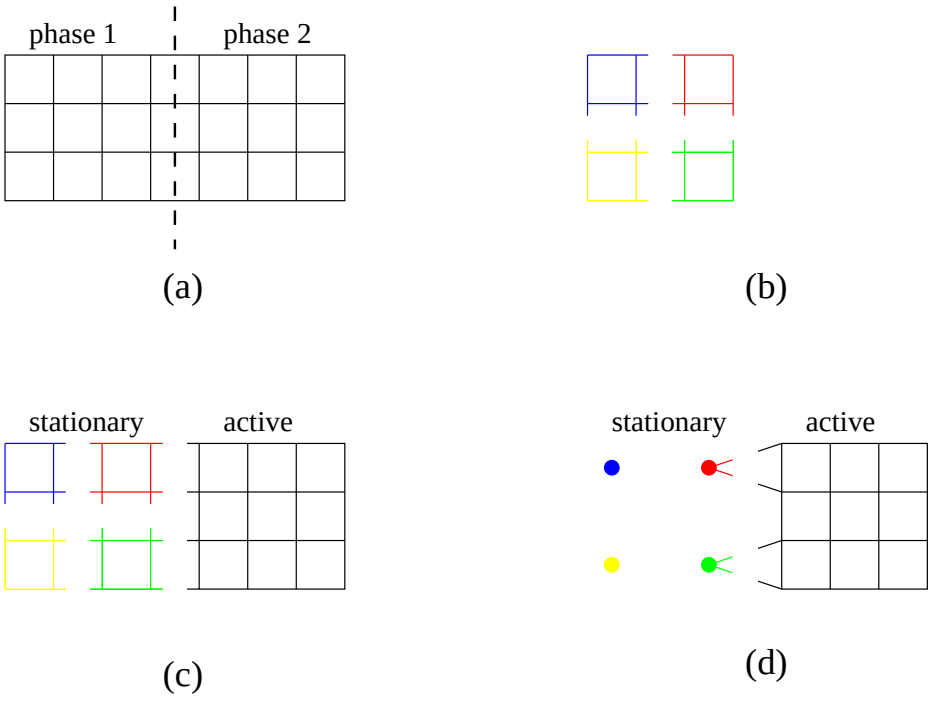

Fig. 2. Multiphase partitioning of a simple two phase mesh: (a) the two phases; (b) the partition of the type 1 vertices; (c) the input graph for the type 2 vertices; (d) the same input graph with stationary vertices condensed

partition the type 1 vertices (each representing a mesh node), shown partitioned in Figure 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 Figure 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. Non-stationary vertices which belong to the current phase are referred to as *active*.

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 Figure 2(d). This can considerably reduce the size of the input graph.

Graph edges. Edges between stationary and active vertices are retained to enhance the interphase data locality, however, as can be seen in Figure 2(d), edges between the condensed stationary vertices are omitted from 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.

Implementation. Although we have illustrated the multiphase partitioning algorithm with a two phase example, the technique can clearly be extended to arbitrary numbers of phases. The multiphase mesh partitioning paradigm then 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, even in parallel, [10]. Furthermore, the modifications required for the partitioner are relatively minor and essentially consist of preventing stationary vertices from migrating. Details can be found in [10].

3 Experimental results

In this section we give illustrative results by testing the multiphase partitioning strategy 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]. Further results for the multiphase scheme on other problem types (such as those which arise when different calculations take place on mesh nodes from those taking place on mesh elements, together with some examples from a real-life contact-impact simulation) can be found in [2, 10].

Table 1. Distinct phase meshes

name	V_1	V_2	E	description
512x256	65536	65536	261376	2D regular grid
crack	4195	6045	30380	2D nodal mesh
dime20	114832	110011	336024	2D dual mesh
64x32x32	32768	32768	191488	3D regular grid
brack2	33079	29556	366559	3D nodal mesh
mesh100	51549	51532	200976	3D dual mesh

The example problems here are constructed by taking a set of 2D & 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 & V_2 represent the number of type 1 & type 2 vertices, respectively, and E is the number of edges. These are possibly the simplest form of two-phase problem and provide a clear demonstration of the need for multiphase mesh partitioning.

The algorithms are all implemented within the partitioning tool JOSTLE[1] and we have tested the meshes with 3 different partitioning variants for 3 different values of P, the number of subdomains/processors. The first of these partitioners is simply JOSTLE's default multilevel partitioning scheme, [8], which takes no account of the different phases and is referred to here as JOSTLE-S. The multiphase version, JOSTLE-M and the parallel multiphase version, PJOSTLE-M, then incorporate the multiphase partitioning paradigm as described here.

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 & λ_2 respectively. These three quality metrics are then averaged for each partitioner and value of P.

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 2 – 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 [8]) 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 & Kumar for their multi-constraint algorithm, [5].

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 seconds for JOSTLE-M

[1] available from `http://www.gre.ac.uk/jostle`

Table 2. Distinct phase results

	$P = 4$			$P = 8$			$P = 16$		
mesh	$\|E_c\|/\|E\|$	λ_1	λ_2	$\|E_c\|/\|E\|$	λ_1	λ_2	$\|E_c\|/\|E\|$	λ_1	λ_2
	JOSTLE-S: jostle single-phase								
512x256	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
64x32x32	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								
512x256	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
64x32x32	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								
512x256	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
64x32x32	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

and 2.22 seconds for JOSTLE-S. For the same mesh in parallel on a Cray T3E (with slower processors) the run times were 5.65 seconds for PJOSTLE-M and 3.27 for PJOSTLE-S (the standard single-phase parallel version described in [9]). On average the JOSTLE-M results were about 1.5 times slower than those of JOSTLE-S and PJOSTLE-M was about 2 times slower than PJOSTLE-S. This is well in line with the 1.5 to 3 times performance degradation suggested for the multi-constraint algorithm, [5].

4 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' mesh partitioner 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 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 however take somewhat longer than the single phase version, typically 1.5-2 times as long although 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 [2], 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.

References

1. C. Bailey, P. Chow, M. Cross, Y. Fryer, and K. A. Pericleous. Multiphysics Modelling of the Metals Casting Process. *Proc. Roy. Soc. London Ser. A*, 452:459–486, 1995.
2. A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, and C. Walshaw. Dynamic Multi-Partitioning for Parallel Finite Element Applications. In E. H. D'Hollander *et al.*, editor, *Parallel Computing: Fundamentals & Applications, Proc. Intl. Conf. ParCo'99, Delft, Netherlands*, pages 259–266. Imperial College Press, London, 2000.
3. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.*, 1:237–267, 1976.
4. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95, San Diego*. ACM Press, New York, NY 10036, 1995.
5. G. Karypis and V. Kumar. Multilevel Algorithms for Multi-Constraint Graph Partitioning. TR 98-019, Dept. Comp. Sci., Univ. Minnesota, Minneapolis, MN 55455, 1998.
6. G. Lonsdale, B. Elsner, J. Clinckemaillie, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Experiences with Industrial Crashworthiness Simulation using the Portable, Message-Passing PAM-CRASH Code. In *High-Performance Computing and Networking (Proc. HPCN'95)*, volume 919 of *LNCS*, pages 856–862. Springer, Berlin, 1995.
7. K. McManus, C. Walshaw, M. Cross, and S. P. Johnson. Unstructured Mesh Computational Mechanics on DM Parallel Platforms. *Z. Angew. Math. Mech.*, 76(S4):109–112, 1996.
8. C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000. (originally published as Univ. Greenwich Tech. Rep. 98/IM/35).

9. C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000. (originally published as Univ. Greenwich Tech. Rep. 99/IM/44).
10. C. Walshaw, M. Cross, and K. McManus. Multiphase Mesh Partitioning. *Appl. Math. Modelling*, 25(2):123–140, 2000. (originally published as Univ. Greenwich Tech. Rep. 99/IM/51).