JOSTLE: parallel multilevel graph-partitioning software – an overview

Chris Walshaw[*] and Mark Cross[**]

[*]*Computing and Mathematical Sciences, University of Greenwich,*
Old Royal Naval College, Greenwich, London SE10 9LS, UK.
URL: staffweb.cms.gre.ac.uk/∼c.walshaw

[**]*School of Engineering, University of Swansea,*
Singleton Park, Swansea SA2 8PP, Wales, UK.

Abstract

In this chapter we look at JOSTLE, the multilevel graph-partitioning software package, and highlight some of the key research issues that it addresses. We first outline the core algorithms and place it in the context of the multilevel refinement paradigm. We then look at issues relating to its use as a tool for parallel processing and, in particular, partitioning in parallel. Since its first release in 1995, JOSTLE has been used for many mesh-based parallel scientific computing applications and so we also outline some enhancements such as multiphase mesh-partitioning, heterogeneous mapping and partitioning to optimise subdomain shape.

Keywords: multilevel refinement; graph-partitioning.

1 Introduction

JOSTLE is a parallel multilevel graph-partitioning software package written at the University of Greenwich. It is freely available for academic and research purposes under a licensing agreement.

In this paper we outline the algorithms devised for and used by JOSTLE, review the results produced, and give a broad overview of the underlying research.

1.1 History

Work on JOSTLE initially started in 1993 when Chris Walshaw joined the Parallel Processing Research Group at the University of Greenwich, with a brief to build a graph-partitioning toolkit. One of the major applications for graph-partitioning is parallel mesh-based computational mechanics codes and so, from its inception, it was anticipated that the software would be able to run in parallel, alongside the solver and include dynamic load-balancing capabilities.

The software was first released into the public-domain in 1995, with the first parallel version appearing in 1997. Since then it has gained a worldwide user community of over 135 licensed sites, and, although some are no longer current, has been licensed by groups based at Los Alamos, Argonne and Sandia National Labs (all in the USA), NASA, and in Universities across the world (including almost all European countries, the USA, Canada, Brazil, Russia, Turkey, Israel, Oman, India, Japan, Singapore and Taiwan). In addition the algorithms have been extended and modified to tackle a variety of partitioning problem variants, some of which are discussed in Section 5.

1.2 The Graph-Partitioning Problem

The k-way graph-partitioning problem (GPP) can be stated as follows: given a graph $G(V, E)$, with vertices V (which can be weighted) and edges (which can also be weighted), partition the vertices into k disjoint sets such that each set contains the same vertex weight and such that the *cut-weight*, the total weight of edges cut by the partition, is minimised. The GPP is usually cast as a combinatorial optimisation problem with the cut-weight as the objective function to be minimised and the balancing of vertex weight acting as a constraint. However, it is quite common to slightly relax this constraint in order to improve the partition quality. It is well known that this problem is NP-complete, [8], so in recent years much attention has been focused on developing suitable heuristics. The GPP has many applications, most notably the partitioning of unstructured meshes for parallel processing (mesh-partitioning), but also including areas such as circuit partitioning, telecommunications, air-traffic management and web search engines.

1.2.1 Recursive bisection

Historically many researchers have approached the GPP by studying its restriction to 2 subsets, the graph-bisection problem. This can then be easily extended to the full problem by recursion, i.e. the graph is bisected into two sub-problems which are themselves bisected to give 4 sub-problems and so on. This technique is known as recursive bisection and has been used with a variety of bisection algorithms, e.g. [20]. It is still widely used and is able to give guarantees on satisfying the balancing constraint, although the resulting partition quality may be limited, [21]. However with the advent of robust k-way partitioners, including JOSTLE, which arguably can be parallelised more easily, and are perhaps better suited to dynamic load-balancing, there is now a considerable body of research on methods which solve the full problem directly, e.g. [10, 14, 29].

1.2.2 Mesh-partitioning

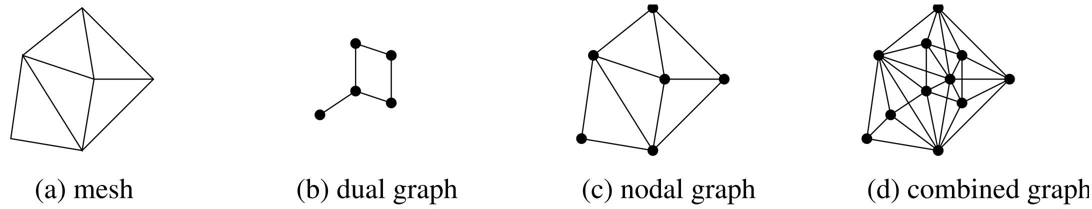


Figure 1: An example mesh and some possible graph representations.

Many of the applications in which partitioning is used involve a parallel computational mechanics simulation, usually 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 and in fact this is a useful abstraction to measure partition quality, even if, as in the case of geometric partitioners, the graph is never explicitly constructed. Thus, if we consider the mesh shown in Figure 1(a), the graph vertices can either represent the mesh elements (the dual graph), Figure 1(b), the mesh nodes (the nodal graph), Figure 1(c), a combination of both (the full or combined graph), Figure 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 neighbouring elements).

1.3 Overview

The rest of this chapter is laid out as follows. First, in Section 2, we outline the key concepts and algorithms and their implementation within JOSTLE. Next, in Section 3, we look at how the multilevel paradigm has contributed to research in the partitioning problem and attempt to explain its runaway success. One of the main applications associated with partitioning, and driving many of the developments in the field, is parallel or distributed computing, and so, in Section 4, we discuss the parallelisation of partitioning schemes. In Section 5 we then outline four extensions which have been successfully applied to address variant partitioning problems. Finally, we summarise the chapter in Section 6.

2 Key algorithms within JOSTLE

In this section we outline the key concepts and algorithms and their implementation within JOSTLE. In particular we look at the the multilevel refinement strategy and outline the refinement algorithm.

2.1 Background

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, [10] and Bui and Jones, [3], and was inspired by Barnard and Simon, [1], who used a multilevel numerical algorithm to speed up spectral partitioning.

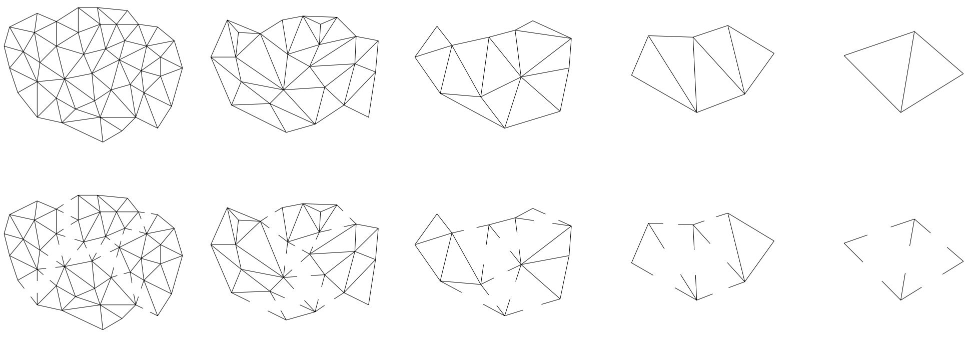


Figure 2: An example of multilevel partitioning

Figure 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 the refinement is only local in nature at each level, a high quality partition is still achieved.

The GPP was the first combinatorial optimisation problem to which the multilevel paradigm was applied and there is now a considerable body of literature about multilevel partitioning algorithms. Initially used as an effective way of speeding up partitioning schemes, it was soon recognised as, more importantly, giving them a ‘global’ perspective, [13], and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (KL), [15], and other optimisation algorithms. Indeed, as we see in §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.

2.2 Multilevel framework

2.2.1 Graph coarsening

A common (although not universal) method to create 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, [10]. 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.

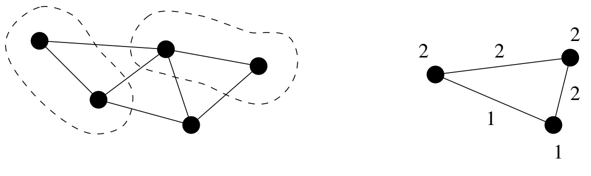


Figure 3: An example of coarsening via matching and contraction

Figure 3 shows an example of this; on the left two pairs of vertices are matched (indicated by dotted lines). 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, matching each unmatched vertex with an unmatched neighbour (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, [13], that it can be beneficial to the optimisation 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 neighbour with the lowest degree (with the aim of trying to avoid highly connected vertices).

2.2.2 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 final partition is balanced, i.e. each subset 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 compute a balanced initial partition (i.e. so that typically $|V_L| \gg k$), [10, 13]. Alternatively, if load-balancing techniques are incorporated alongside the refinement algorithm, as is the case with JOSTLE, [29], the contraction can be terminated when the number of vertices in the coarsest graph is the same as the number of subsets required, k, and then vertex i is assigned to subset S_i.

2.2.3 Partition extension

Extension is the reverse of coarsening and operates in tandem with the refinement algorithms. Thus, 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 subset S_i then the matched pair of vertices that it represents, $v_1, v_2 \in V_l$, are also assigned to S_i.

2.3 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 optimisation, 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 implementations exist for simulated annealing, tabu search, genetic algorithms, cooperative search and even ant colony optimisation (see [27] for references).

2.3.1 The gain and preference functions

Two key concepts for refinement (which we will use later in the chapter to extend and adapt the algorithms) are the ideas of *gain* and *preference*. The gain, $\texttt{gain}(v, q)$, of a vertex v in subdomain S_p can be calculated for every other subdomain, S_q, $q \neq p$, and expresses some measure of how much the partition would be improved, were v to migrate to S_q. The preference, $\texttt{pref}(v)$, is then just the preferred subdomain for v to migrate to, and thus the value of q which maximises the gain; i.e. if $\texttt{gain}(v, q)$ attains $\max_r \texttt{gain}(v, r)$, then $\texttt{pref}(v) = q$. In the event of a tie, other criteria, such as subdomain weight, are used to make the final preference decision.

For the classic GPP then, the $\texttt{gain}(v, q)$ just expresses the reduction in the cut-weight, although the idea of gains is generic (see §5.1 and §5.2 for different examples). Also, since there can never be a reduction in cut-weight if a vertex v is transferred to a subdomain S_q to which it is not adjacent (because there will be no cut edges between v and S_q), gains are only calculated for *border* vertices (i.e. those actually adjacent to another subdomain) and furthermore, it usually suffices to calculate gains just for the adjacent subdomains (although see §5.2). This in turn restricts the preference to adjacent subdomains and indeed, in a high quality partition of a sparse graph, most border vertices will only be adjacent to one other subdomain.

2.3.2 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 (i.e. $\texttt{gain}(v, q) > 0$) without exceeding the allowed imbalance; or (b) the move improves the balance without changing the cost ($\texttt{gain}(v, q) = 0$). Although this scheme cannot guarantee perfect balancing, it has been applied to very good effect, [14], and is extremely fast.

Although not the default behaviour, JOSTLE includes a greedy refinement scheme, accessed by turning off the hill-climbing abilities of the optimisation (see §2.3.3).

2.3.3 The k-way Kernighan-Lin algorithm

A more sophisticated class of refinement method is based on the Kernighan-Lin (KL) bisection optimisation algorithm, [15], which includes a degree of hill-climbing ability to enable it to escape from local minima. This has been extended to k-way partitioning in different ways by several authors (e.g. [10, 14, 29]) and recent implementations almost universally use the linear time complexity improvements (e.g. bucket sorting of vertices) introduced to partitioning by Fiduccia and Mattheyses, [6].

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 initialised by calculating the *gain* – the potential improvement in the cost function (see

§2.3.1) – 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 neighbours 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 subsets 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, (see below, §3.1). Note that if $\lambda = 0$ then the refinement is purely greedy in nature. Once the inner loop is terminated, any vertices remaining in the list (vertices whose transfer has not been confirmed) are transferred back to the subsets they came from when the best cost was attained.

JOSTLE uses just such a refinement algorithm, [29], 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, [11], 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, [29].

3 Multilevel context

One of the most important factors which has allowed JOSTLE to tackle a range of different partitioning problems (including successful parallel partitioning, Section 4 and extensions such as those described in Section 5) is the multilevel framework. Indeed, its success as a paradigm has inspired further investigation in a variety of fields including the travelling salesman problem, the graph colouring problem, the facility location problem and force-directed graph layout, as well as research into the dynamics of the framework itself (see [27] for individual references). In this section we look at how the multilevel paradigm has contributed to research in the partitioning problem and attempt to explain its runaway success.

3.1 Multilevel refinement: typical results

To illustrate the potential gains that the multilevel paradigm can offer, we give some example results. These are not meant to be exhaustive in any way but merely give an indication of typical performance behaviour.

3.1.1 Asymptotic tests

In [27], detailed tests are carried out to assess the impact of multilevel refinement on the GPP. Here we summarise those results.

The experimental data consists of two test suites, one of which is a smallish collection of 16 sparse, mostly mesh-based graphs, drawn from a number of real-life applications, and often used for benchmarking. The other test suite consists of 90 instances compiled to test graph-colouring algorithms and including a number of randomly generated examples. Although perhaps not representative of partitioning applications, they reveal some interesting results. This colouring test suite is further subdivided into 3 density classes; low (under 33%) with 58 out of 90 instances, medium (between 33% and 67%) with 23 instances and high (over 67%) with just 9 instances.

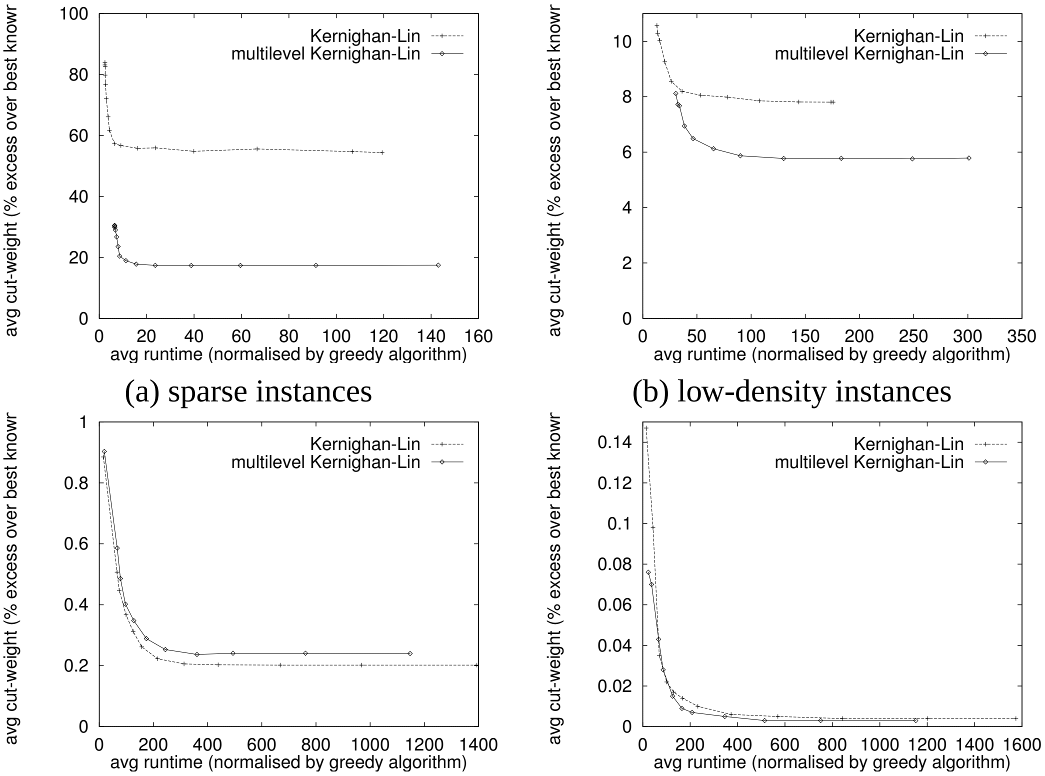
In this context, the density, or *edge density*, of a graph, $G(V, E)$, is defined as the percentage of all possible edges and given by $2|E|/[|V| \cdot (|V| - 1)]$, so that a *complete* graph (where every vertex is adjacent to every other), with $|V| \cdot (|V| - 1)/2$ edges, has a density of 100%. Although the distinction between sparse and low-density graphs is not always clear, especially for small examples, typically we use *sparse* to mean families of graphs for which the number of edges $|E|$ is $O(|V|)$ and so the density decreases with increasing $|V|$. On the other hand, *low-density* tends to mean families of graphs which have $O(|V|^2)$ edges but for which the density, remains constant with increasing $|V|$ (for example the colouring test suite contains a series of randomly generated graphs of different sizes but fixed density of around 0.1).

The tests compare the JOSTLE implementation of the Kernighan-Lin (KL) algorithm against its multilevel counterpart (MLKL). As mentioned in §2.3.3 and similar to most local search schemes, the KL algorithm contains a parameter, λ, known as the *intensity*, which allows the user to specify how long the search should continue before giving up.

To assess the algorithms, we measure the run-time and solution quality for a chosen group of problem instances and for a variety of intensities. We then normalise these values with reference solution quality and run-time values and finally plot averaged normalised solution quality against averaged normalised run-time for each intensity value.

Figure 4(a) shows the results for the sparse suite and the dramatic improvement in quality imparted by the multilevel framework is immediately clear. Even for purely greedy refinement (i.e. the extreme left-hand point on either curve where $\lambda = 0$ – see above §2.3) the MLKL solution quality is far better than KL and it is results like these that have helped to promote multilevel partitioning algorithms to the status they enjoy today.

Figures 4(b)-(d) meanwhile show the partitioning results for the colouring test suite. Figure 4(b) more or less confirms the conclusions for the sparse results and although the curves are closer together, MLKL is the clear winner. For the medium



(a) sparse instances

(b) low-density instances

(c) medium-density instances

(d) high-density instances

Figure 4: Plots of convergence behaviour for the partitioning test suites

and high-density examples however, it is a surprise (especially considering the widely accepted success of multilevel partitioning) to find that these conclusions are no longer valid. For the high-density instances, Figure 4(d), MLKL is still the leading algorithm, although only very marginally. However for the medium-density results, Figure 4(c), MLKL fails to achieve the same performance as KL and the multilevel framework appears to actually hinder the optimisation. We discuss this further in the following section.

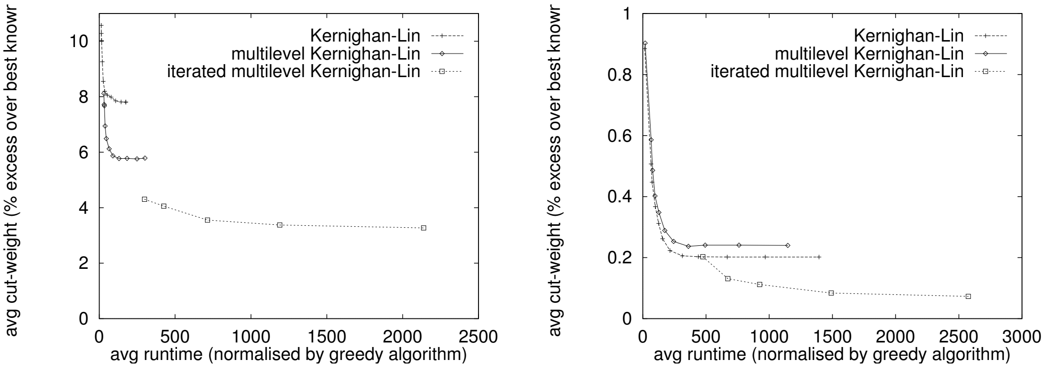
3.1.2 Iterated multilevel partitioning

Although the medium density results are disappointing, in fact a simple resolution does exist which works by reusing the best partitions that have been found. Indeed, given any partition of the original problem we can carry out *solution-based coarsening* by insisting that, at each level, every vertex v matches with a neighbouring vertex in the same set. When no further coarsening is possible this will result in a partition of the coarsest graph with the same cost as the initial partition of the original. Provided the refinement algorithms guarantee not to find a worse partition than the initial one, the multilevel refinement can then guarantee to find a new partition that is no worse than the initial one.

This sort of technique is frequently used in graph-partitioning for dynamic load-balancing, e.g. [19, 28], although if the initial partition is unbalanced, the quality guarantee can be lost in satisfying the balance constraint. However it can also be used to find very high quality partitions, albeit at some expense, and the multilevel procedure can be iterated via repeated coarsening and uncoarsening. At each iteration the current best solution is used to construct a new hierarchy of graphs, via solution-based coarsening, and, as we have seen, the process guarantees not to find a worse solution than the initial one. However, if the matching includes a random factor, each iteration is very likely to give a different hierarchy of graphs to previous iterations and hence allows the refinement algorithm to visit different solutions in the search space.

We refer to this process as an *iterated multilevel algorithm* (see also [24] for a variation of this technique). Note that it requires the user to specify a different intensity parameter, γ, namely the number of failed outer iterations (i.e. the number of times the algorithm coarsens and uncoarsens the graph without finding a better solution). We can then vary γ, the outer intensity parameter, to test the iterated multilevel algorithm and below we give some sample results for this scheme (which has also been used to find very high quality partitions for benchmarking purposes).

Figure 5 illustrates the results for the iterated multilevel algorithm (IMLKL) alongside the MLKL and KL results for low and medium-density subclasses of the colouring suite. These plots contain the same information about MLKL and KL as Figures 4(b) and 4(c), only here it is more compressed because of the long IMLKL runtimes.



(a) low-density instances

(b) medium-density instances

Figure 5: Plots of convergence behaviour including iterated multilevel partitioning results

We do not show results for the sparse and high-density instances because they are not so interesting; for the sparse suite IMLKL more or less continues the MLKL curve in Figure 4(a) with a few percentage points improvement and very shallow decay whilst for the high-density instances IMLKL does not appear to offer much improvement at all. However for the low and medium-density subclasses, in Figures 5(a) and 5(b) respectively, the asymptotic performance offered by IMLKL is impressive and worthy of further and more thorough investigation. In both cases IMLKL dramatically improves on MLKL and, for the medium-density instances, even appears to overcome the shortcomings of MLKL and exceeds the KL results.

3.2 Multilevel landscapes

It is of great interest to ask how the multilevel paradigm helps to solve partitioning and other combinatorial problems. In this section we attempt to give an insight into the dynamics of the multilevel method by considering characteristics of the typical results and, in particular, of the hierarchy of solution spaces produced by the coarsening.

As mentioned above, §2.2, the coarsening constructs a series of approximations to the original problem; it is hoped that each problem P_l retains the important features of its parent P_{l-1} but the (usually) randomised and irregular nature of the coarsening tends to preclude any rigorous analysis of this process.

On the other hand, viewing the multilevel process from the point of view of the objective function and, in particular the hierarchy of solution spaces, is considerably more enlightening. Typically, the coarsening is carried out by matching groups pairs of solution variables together and representing each pair with a single variable in the coarsened space. Previously authors have made a case for multilevel refinement (and partitioning in particular) on the basis that the coarsening successively ***approximates*** the problem. In fact it is somewhat better than this: as discussed in [27] and illustrated below, the effect, if carried out correctly, is to ***filter*** the solution space by placing restrictions on which solutions the refinement algorithm can visit.

To see this suppose that two vertices $v_1, v_2 \in G_l$ are matched and coalesced into a single vertex $v \in G_{l+1}$. When a refinement algorithm is subsequently used on G_{l+1} and whenever v is assigned to a subset, both v_1 and v_2 are also being assigned to that subset. In this way the matching restricts a refinement algorithm working on G_{l+1} to consider only those configurations in the solution space in which v_1 and v_2 lie in the *same* subset, although the particular subset to which they are assigned is not specified at the time of coarsening. Since many vertex pairs are generally coalesced from all parts of G_l to form G_{l+1} this set of restrictions is equivalent to filtering the solution space and hence the surface of the objective function.

We can then conjecture that, if the coarsening manages to filter the solution space so as to gradually ***smooth*** the objective function, the multilevel representation of the problem combined with an iterative refinement algorithm should work well as an optimisation metaheuristic. In other words, by filtering a large amount of irrelevant detail from the solution space (in particular the higher cost solutions which are not close to local optima), the multilevel component allows the refinement algorithm to find regions of the solution space where the objective function has a low average value (e.g. broad valleys). On a more pragmatic level this same process also allows the refinement to take larger steps around the solution space (e.g. rather than swapping single vertices, the local search algorithm can swap whole sets of vertices as represented by a single coarsened vertex).

3.2.1 Motivating example

Figure 6 shows an example of how this process might work for a simple objective function (see [26] for details). On the left hand side the objective function is filtered and smoothed by selecting and removing possible solutions (essentially at random). The initial solution for the final coarsened space (shown in the bottom right hand figure) is then trivial (because there is only one possible state) although the resulting solution is far from optimal for the overall problem. However this state is used as an initial configuration for the next level up and (in this case) a *steepest descent* refinement policy finds the nearest local minimum (steepest descent refinement will only move to a neighbouring solution if the value of the objective function is lower there). Here the vertical lines indicate the solutions visited whilst the arrows above show the trajectory through the solution space. Repeated application of this process on the increasingly finer levels continues to improve the best-found solution. Finally this gives a good starting point for the original problem and (in this case) the optimal solution can be found (although the arrows are indistinct, the vertical lines show the solutions visited).

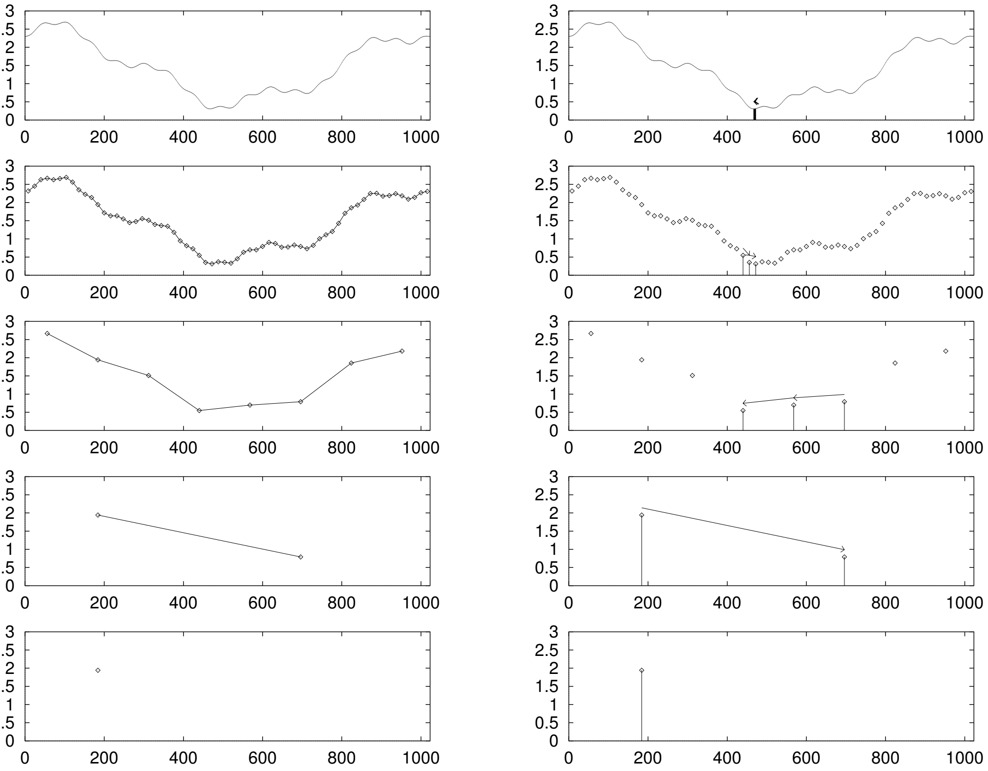


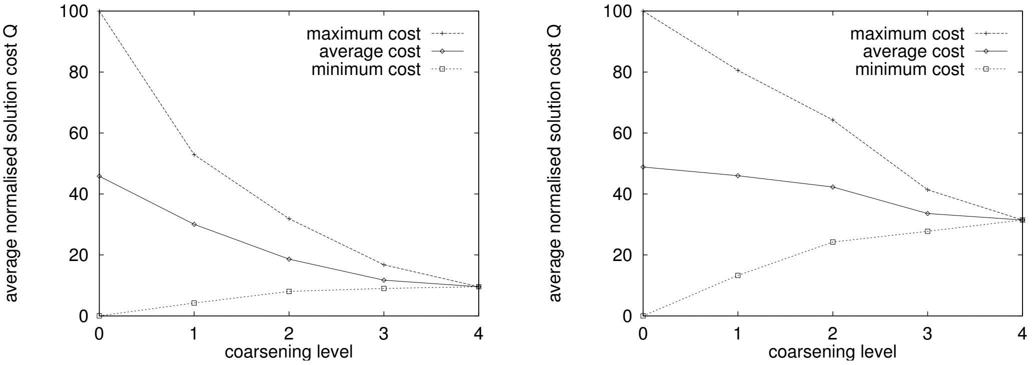
Figure 6: The multilevel scheme applied to a simple objective function

Of course this motivational example might be considered unrealistic (in particular an objective function cannot normally be pictured in 2D). However, consider other heuristics, such as repeated random starts combined with steepest descent local search, or even simulated annealing; without lucky initial guesses either might require hundreds of iterations to find the optimal solution of this trivial problem.

3.2.2 Experimental results

The conjecture that the multilevel framework filters out high cost solutions (partitions) from the solution space is validated in [35], but we give an outline of the results here. The idea is to enumerate all possible partitions of a given graph at each level of the coarsening and to see how the coarsening affects the solution landscape.

Clearly it is only possible to enumerate every possible solution for very small graphs, since the number of solutions is given by ${}^{(|V|-1)}C_{|V|/2}$ (where ${}^{n}C_{r}$ denotes the number of combinations of r objects from n and is given by $n!/r!(n-r)!$). Thus, for a graph of size $|V| = 32$ the total number of solutions is ${}^{31}C_{16} = 300,540,195$, and the enumeration code, running on a 1 GHz Pentium processor, was able to evaluate every solution in approximately 10 to 40 minutes, depending on the edge density of the example. To illustrate the exponential size of the partitioning problem, this means that for graphs of size $|V| = 64$, a similar experiment would have taken approximately $1,000$ and $4,000$ years to run! As an alternative then, we also look at some larger graphs with $|V| = 1,024$ where the solution space is sampled randomly.



(a) sparse instances

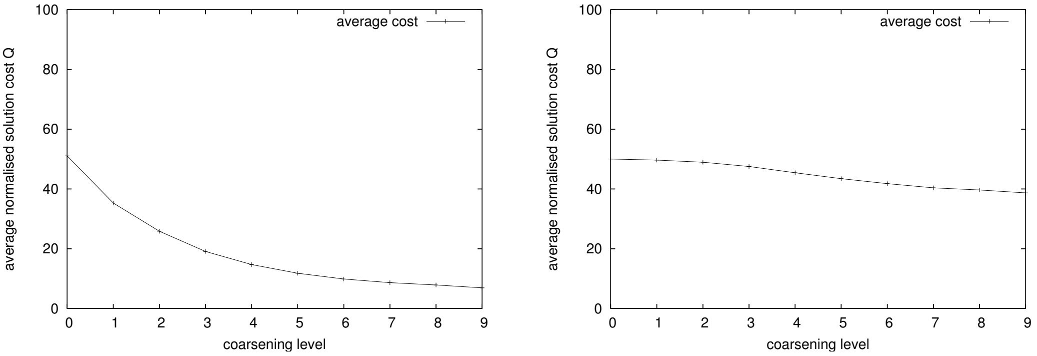
(b) high-density instances

Figure 7: Enumeration results for the small GPP instances

Enumeration results for the small graphs ($|V| = 32$) are given in Figure 7, although here we only show sparse and high-density results (the low and medium results are intermediate and similar). For each plot we take the maximum, average and minimum values of Q, the normalised solution quality, for each graph and each level and then average these values over the 16 graphs in each subclasses (sparse and high-density). These averaged normalised values are then plotted against the coarsening level.

It is perhaps not immediately obvious what conclusions to draw from these plots, but we interpret the results as follows. Firstly, in all four density subclasses, the average value of the cost function decreases as the coarsening progresses (i.e. as the level increases). In other words the coarsening is filtering out the higher cost solutions at a greater rate than the lower cost ones. Ideally we would like it to avoid filtering out any low cost or optimal solutions (although this would mean that the multilevel scheme would find the optimal solution at the end of the coarsening phase, which seems a little too much to hope for). However, an important point to note is that the size of the coarsest spaces is relatively small. Thus we might reasonably expect all but the most basic of refinement schemes (i.e. all those with the ability to escape local minima) to be able to find the best solution available at level 2 (which only has 35 possible solutions). This means that the refinement should already have a good initial solution for levels 1 and subsequently 0.

Furthermore, with regard to the effects of density, the observations above (§3.1), do seem to be borne out. Thus the more dense the problem, the shallower the gradient of the average cost curves. In other words, in higher-density graphs the multilevel coarsening filters out almost as many low cost solutions as high cost and hence is less effective.



(a) sparse instances

(b) high-density instances

Figure 8: Sampling results for the large GPP instances

In Figure 8 results for somewhat larger graphs ($|V| = 1,024$) are given. Here the solution space is just sampled and hence we have no values for the minimum or maximum cost of solutions over the whole space. Once again the fact that the average value of the cost function always decreases for each subclass demonstrates that the coarsening is acting as an effective filter. However, now the effects are much more dramatic and one instantly see the success of the multilevel technique for sparse examples, where the high cost solutions are almost completely filtered out by the final coarsening. And, once again, as for the small examples, increasing graph density is seen to have an adverse effect on the success of the filtration.

In summary then, it seems that the multilevel framework works by filtering out high cost solutions from the solution space. However, as graph density increases it is harder for the coarsening algorithms to find them. Fortunately, however, most partitioning applications feature sparse graphs.

4 Parallel partitioning

4.1 Background

One of the main applications associated with the GPP, and driving many of the developments in the field, is parallel or distributed computing, particularly with the purpose of enabling large-scale mesh-based simulations in computational science. In these cases, the graph vertices represent units of computational workload whilst the edges represent data dependencies. For optimal parallel efficiency, it is typical that the parallel processors each need the same amount of computational work, thus inducing the load-balancing constraint. Meanwhile, the minimisation of interprocessor communications is modelled by the cut-weight objective function (although, as several authors have pointed out, e.g. [9], since this measures the total communication *volume*, it is not necessarily the best representation of communications overhead).

4.1.1 Requirements analysis

Such mesh-partitioning problems usually arise in one of three different ways and can be characterised as the:–

(i) **static partitioning problem** (the classical problem) which arises in trying to distribute an existing mesh amongst a set of processors;

(ii) **static load-balancing problem** which arises from a mesh that has been generated in parallel (and thus is already distributed);

(iii) **dynamic load-balancing/partitioning problem** which arises either from adaptively refined meshes, or from meshes in which the computational workload for each mesh entity can vary with time, or even from machines on which (due to external user load) the computational resources may vary.

In the last two cases, (ii) and (iii), the initial data is a distributed mesh which may be neither load-balanced nor optimally partitioned. One way of dealing with this is to ship the mesh back to some host processor, run a serial partitioning algorithm on it and redistribute. However, this is unattractive for many reasons. Firstly, an $O(N)$ overhead for the mesh-partitioning is simply not scalable if the solver is running at $O(N/P)$, where here N is the number of graph vertices and P the number of processors. Indeed the mesh may not even fit into the memory of the host machine and thus incur enormous delays through memory paging. In addition, a partition of the mesh (which may even be optimal) already exists, so it makes sense to reuse this as a starting point for repartitioning, [19, 33]. In fact, not only is the load-balancing likely to be unnecessarily computationally expensive if it fails to use this information, but also the mesh elements will be redistributed without any reference to their previous 'home processor' and heavy data migration may result. Thus, because the mesh is already distributed, it is a natural strategy to repartition it *in situ*.

It could be argued that case (i) is best handled by a serial partitioning algorithm. However, once again an $O(N)$ start-up cost for the mesh-partitioning may not be acceptable and the same memory problems can arise. Also, assuming that a parallel machine is available to run the solver, it makes sense to use it for the initial partition as well.

With these issues in mind, JOSTLE is implemented to provide a partitioning framework that:–

(a) works in parallel;

(b) can optimise an existing partition;

(c) can find a high quality partition *independent* of the existing partition;

(d) incorporates load-balancing techniques.

4.1.2 The parallel partitioning paradox

This raises a paradox for case (i): we seek to compute, in parallel, a partition of a previously unpartitioned mesh; however, to do it in parallel we must first distribute the mesh sensibly amongst the processors and to distribute the mesh sensibly we must first find a reasonable partition.

In fact it is requirement (c) that, if met, can answer this paradox. It is easy to distribute a mesh if there are no guarantees about the partition quality (e.g. by assigning mesh entities to processors on a cyclic basis). Thus if requirement (c) can be met, we can indeed solve problem (i) to high quality by initially using a crude distribution of the mesh and then optimising it in parallel.

In this section we outline the JOSTLE framework which satisfies all four requirements (a)-(d) and which thus aims to solve all three problems (i)-(iii), using the same algorithm. In particular, we will focus on the solution of the static partitioning problem (i) and the requirements (a)-(c). For more details of the dynamic aspects, and a new self-adapting dynamic strategy, see [28], where a dynamic partitioning scheme is described that trades off the minimisation of cut-weight against the changes to the partition (if the partition is left relatively unchanged, most of the data can stay in place and does not need to migrate to another processor).

4.2 Parallel multilevel partitioning

The multilevel scheme described in Section 2 has been parallelised within JOSTLE, [30]. The communications are performed via the Message Passing Interface library MPI, and the implementation uses the owner-computes single-program multiple-data paradigm. Thus the vertices in each subdomain, S_p, are assigned to processor p, which also holds a one deep halo or read-only copy of vertices adjacent to S_p.

4.2.1 Parallelising the multilevel framework

In fact, most of the multilevel framework, including the coarsening and expansion, is inherently localised and hence relatively easy to implement in parallel, although with some minor differences. Most notably, the graph is already distributed and an initial partition already exists, and so does not need to be computed for the coarsest graph. However, the coarsening algorithms can work almost unchanged, although some migration of vertices is allowed to facilitate matching of vertices in two different subdomains (this feature can also be switched off by the user at run-time to reduce vertex migration for dynamic load-balancing).

As mentioned in §2.2, the coarsening usually continues until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and this gives us automatically an initial partition with one vertex per subdomain. However, although coarsening down to a single vertex per subdomain is rapid in serial (since at the coarsest levels, the graphs become very small indeed), in parallel it can be relatively inefficient since each coarsening stage involves several communication calls. For this reason, once the size of the graph falls below a given threshold, each processor broadcasts its portion so that every processor has a copy of the entire graph. The coarsening and expansion/refinement process can then continue entirely in serial, using the algorithms described in §2.2 and §2.3, with every processor duplicating the work. The optimum threshold at which to construct the global graph is of course machine dependent (based on the ratio of the cost of communication and computation) but the default setting (which can be reset at run-time) is 20 vertices per processor. Indeed, for maximum scalability it may make sense to not to the construct the global graph at all and this option can also be chosen at run-time.

4.2.2 Parallelising the refinement algorithm

As described in §2.2, a typical serial Kernighan-Lin (KL) type algorithm consists of inner and outer iterative loops. The inner loop picks vertices (usually those with the highest gain) and migrates them from one subdomain to another. It will not usually visit any vertex more than once during the course of an inner loop, in order to prevent cyclic behaviour, and terminates when all vertices have been visited, or when there is little prospect of further improvement with the unvisited vertices. The outer loop is simply repeated applications of the inner loop and terminates when no migration takes place within an inner loop.

The main problems in parallelising this procedure lie within the inner loop. Firstly, if the graph is distributed, migrating one vertex at a time involves far too much communication overhead (with most of the processors lying idle most of the time) and for this reason we employ a bulk migration scheme where each processor finds as many border vertices as possible to migrate and moves them once per iteration of the outer loop.

The second, and more difficult problem lies in determining which vertices to migrate. In fact, the swapping of vertices between two subdomains is an inherently non-parallel operation and hence there are some difficulties in arriving at efficient parallel versions, [18]. Since all the processors are acting in parallel on the vertices that they own, simply moving vertices with the highest gain is not a satisfactory solution as it means that adjacent vertices may be swapped simultaneously (a non-optimal event known as a *collision*) and this may lead to an *increase* in the cost.

An important part of our strategy for tackling the problem of collisions is to note that every border vertex in a subdomain, S_p say, has a preferred subdomain for it to migrate to expressed by the preference function (§2.3.1). It is therefore possible to fix border regions, temporarily at least, and define subsets $B_{pq} = \{v \in S_p : \texttt{pref}(v) = q\}$; in other words, B_{pq} is the set of vertices in the border of subdomain S_p with a preference to migrate to S_q. We refer to these sets as *subdomain faces*.

Each pair of subdomain faces, $B_{pq} \bigcup B_{qp}$ then forms an *interface* region I_{pq}. Since the preference of every border vertex is fixed throughout each outer iteration (because it is only determined once during the iteration), these interfaces cannot change during that iteration. This allows us to isolate regions of the graph, which in turn helps to avoid collisions.

Three different refinement strategies, implemented in JOSTLE, are described in [30]. Each chooses, in parallel, vertices to migrate, whilst attempting to avoid collisions. In summary they are:–

- **Interface Optimisation**. A serial optimisation algorithm is executed independently in each of the interface regions I_{pq} by one or other of the processors p and q.

- **Alternating Optimisation**. One of each pair of subdomain faces, B_{pq}, B_{qp}, is selected and the owning processor chooses vertices from that face for migration (to its opposite face). A certain amount of imbalance is crucial for this algorithm to work because the active processor is not allowed to create serious imbalance and so if the tolerance is zero no vertices can be migrated. In the following iteration of the outer loop the alternate face is selected.

- **Relative Gain Optimisation**. If we think of the gain as a force or potential we can imagine a relative gain for every border vertex, v, according to the neighbouring vertices in the opposite face. Intuitively, if the gain of the opposite vertices is high they are likely to migrate and so v should not migrate; if the opposing gain is low then there is little danger of a collision if v migrates and so the relative gain attempts to express this migration potential. Each processor then picks an 'appropriate' weight of vertices to migrate, highest relative gain first. The fact that the gains of all vertices in the opposite face are taken into account (in the relative gain calculation) helps to avoid most collisions.

In [30] these three approaches are compared. To summarise, the interface optimisation algorithm is very rapid and generally produces the best results in terms of cut-weight. However, it does not completely remove imbalance in the final partition and a hybrid algorithm, using relative gain with a final clean-up step of interface optimisation, produces very similar results (about 2.5% worse), equally rapidly *and* removes most of the imbalance. The results are also compared with another partitioning tool, ParMETIS, [12], and shown to be of higher quality (about 11% better) although taking longer to compute.

4.3 Typical results

The parallel version of JOSTLE has been tested and deployed on a variety of parallel machines, including the ASCI Red at Sandia National Laboratories, the world's first TeraFLOPS supercomputer. Here we give some typical results from a Cray T3E-900/512 at the University of Stuttgart.

For each test the mesh is read in parallel and distributed contiguously to the processors (i.e. processor 0 is given the first $|V|/P$ vertices, processor 1 the next $|V|/P$, etc.). This means the initial partition can be of extremely poor quality (although see §4.3.2 for results on the impact of the initial distribution). The algorithm is allowed a 5% final imbalance tolerance (which can be reset at run-time by the user).

4.3.1 Run-time and serial/parallel comparisons

			$P = 16$				$P = 64$			
mesh	$\lvert V\rvert$	$\lvert E\rvert$	$t_p(s)$	C_p	C_s	$\frac{C_s}{C_p}$	$t_p(s)$	C_p	C_s	$\frac{C_s}{C_p}$
4elt	15606	45878	0.49	1070	993	0.93	0.84	2728	2707	0.99
t60k-d	60005	89440	0.54	925	952	1.03	0.70	2381	2412	1.01
t60k-n	30570	90575	0.87	1753	1817	1.04	0.79	4378	4379	1.00
dime20	224843	336024	1.49	1305	1257	0.96	1.26	3632	3665	1.01
mesh100	103081	200976	2.85	4662	4420	0.95	2.61	9993	10478	1.05
t60k-f	90575	360030	3.46	5190	4731	0.91	2.87	12118	12020	0.99
fe-ocean	143437	409593	6.52	8546	8904	1.04	3.60	21845	22867	1.05
fe-rotor	99617	662431	8.36	22789	22050	0.97	6.58	50580	52764	1.04
cyl3	232362	457853	12.32	9976	10543	1.06	6.34	20211	21014	1.04
598a	110971	741934	17.17	27009	28198	1.04	10.38	59866	60775	1.02
average						0.99				1.02

Table 1: The results of the parallel interface algorithm showing parallel run-time in seconds $t_p(s)$, the parallel and serial cut-weight, C_p and C_s respectively, and their ratio, C_s/C_p.

The results of the parallel multilevel partitioning using the interface optimisation algorithm are shown in Table 1 for two values of P (the number of processors). It shows the parallel run-time in seconds, $t_p(s)$, the cut-weight found by the parallel and serial algorithms, C_p and C_s respectively, and the ratio of these two figures. The results are sorted in order of the run-time when $P = 16$ and indicate that it is roughly dependent on $|V|+|E|$ (although the mesh numbering can also play an important role here – see below §4.3.2).

The ratio shows how well the parallel version performs. Thus the value 1.04 (t60k-n, $P = 16$) means that the serial partitioning resulted in a cut-weight 1.04 times as large (or 4% larger) than that of the parallel partitioning. Indeed, as can be seen, the serial results are very similar to the parallel ones, with a maximum of 9% difference (t60k-f, $P = 16$). The average difference in the quality ranges between 1% better and 2% worse over the different values of P with an overall average of under 1% depreciation for the serial results. This demonstrates that the parallel partitioner produces results of more or less the same quality as the serial partitioner, an impressive result considering the fact that the serial version has access to all of the data whilst a processor running the parallel code can only access its local portion of the graph.

The timing results also indicate parallel performance. Achieving high parallel speedup within partitioning codes such as JOSTLE is not as easy as, say, a typical CFD or CM code. For a start the algorithms use only integer operations and so there are no megaFLOPS to 'hide behind'. In addition, most of the work is carried out on the subdomain boundaries and so very little of the actual graph is used. Also the partitioner itself may not necessarily be well load-balanced and the communications cost may dominate on the coarsest reduced graphs since at this stage there are very few vertices per processor. However, the timings generally decrease as P increases (except on the smaller meshes where there is so little computational work that these figures mainly show parallel communication overhead), indicating good performance for this sort of code. Furthermore, as was explained in §4.1.2, partitioning on the host may be impossible or at least much more expensive and if the cost of partitioning is regarded (as it should be) as a parallel overhead, it is usually extremely inexpensive relative to the overall solution time of the problem.

4.3.2 The impact of the initial distribution

	initial	$P = 16$			$P = 64$		
mesh	distribution	C_0	C	$t(s)$	C_0	C	$t(s)$
t60k-n	cyclic	86929	1821	2.08	89586	4476	1.45
t60k-n	random	84843	1737	2.06	89103	4385	1.43
t60k-n	block	6639	1753	0.88	10536	4378	0.80
t60k-n	greedy	2248	1758	0.44	5336	4476	0.61
cyl3	cyclic	432639	10299	14.71	451608	20564	6.86
cyl3	random	429109	10195	14.87	450678	20606	6.83
cyl3	block	351188	9976	12.31	388139	20211	6.34
cyl3	greedy	20014	10398	3.49	37442	20911	3.47

Table 2: Results showing the effect of different initial distributions (with cut-weight C_0) on the final partition quality (cut-weight C) and the parallel partitioning time, t.

We have suggested the requirement in §4.1.1 that the partitioner should be able to find a high quality partition *independent* of the existing partition. It is of interest to ask, therefore, whether this is possible and indeed what impact the initial distribution has on the outcome of the final partition. In Table 2 we compare four different initial distribution schemes for two example meshes. The cyclic distribution assigns vertex i to processor p if i mod $P = p$, i.e. vertex numbers $0, P, 2P, \ldots$ are given to processor 0, vertices $1, P + 1, 2P + 1, \ldots$ to processor 1, etc. The random distribution assigns them randomly. The block distribution is the one used for the results in Table 1 and assigns the first $|V|/P$ vertices to processor 0, etc., while the greedy algorithm is a (serial) graph-based implementation of Farhat's algorithm, [4]. Note that the cyclic, random and block distributions are all parallel input algorithms in the sense that the mesh can be read in from file in parallel, while the greedy algorithm requires the execution of a separate serial partitioner. The results show for each value of P the cut-weight of the initial distribution, C_0, the cut-weight of the final partition, C and the partitioning time in seconds, $t(s)$.

The results clearly demonstrate two things. Firstly, modulo a certain amount of 'noise' (inevitable for randomised discrete optimisation algorithms such as these) with a maximum variation of 4.8% in the final cut-weight, the quality of the final partition is independent of the quality of the initial distribution. Thus the partitioning techniques are clearly seen to provide global rather than just local optimisation. Secondly, the partitioning time is *strongly* dependent on the initial distribution, with the poorly distributed results taking much longer to partition.

Regarding the initial distribution schemes, note that the block distribution can lead to a wide variation in initial cut-weight dependent on whether the mesh has been numbered with some form of structure (i.e. as in t60k-n, vertices which are close in index have a good chance of being neighbours in the graph) or not (i.e. as in cyl3, where no such relation appears to exist). Finally note that the cyclic scheme almost always (and always in Table 2) produces an initial cut-weight worse than the random distribution for precisely the opposite reason; even if such a relation exists in the numbering it is destroyed by placing contiguous vertices on different processors.

5 Variants and extensions

JOSTLE has been successfully extended to a range of partitioning problems variants and in this section we outline four modifications to the basic algorithms described in Section 2. In particular, in §5.1 we show how the aspect ratio (shape) of the subdomains can be optimised as an alternative to the cut-weight. Meanwhile, in §5.2 we look at mapping graphs onto parallel computers with heterogeneous communications links – a successful mapping is then one in which adjacent subsets generally lie on 'adjacent' processors. In both cases, the coarsening and refinement schemes require relatively few modifications and the new cost function is optimised solely by simple changes to the gain function (§2.3.1) and, in the case of aspect ratio optimisation, the matching. This makes an interesting point: using the multilevel framework, the ***global*** layout of the final partition can be radically changed just by modifying the ***local*** cost function. This corroborates the evidence in Section 3 that the multilevel framework adds some sort of global perspective to partitioning schemes.

Meanwhile, in §5.3 and §5.4, two variants are described which essentially use the standard multilevel partitioner as a black-box solver and build a framework around it. In particular, we look at multiphase partitioning problems in §5.3, and in §5.4 discuss the use of evolutionary search techniques to find very high quality partitions.

5.1 Optimising subdomain aspect ratio (shape)

In [32] Walshaw and Cross describe a variant of JOSTLE which optimises the aspect ratio of the subdomains, rather than the cut-weight. We outline those ideas here.

The need for aspect ratio (AR) optimisation arises from a class of mesh-based solution techniques. A natural *parallel* solution strategy for the underlying problem is to use an iterative solver such as the conjugate gradient (CG) algorithm together with domain decomposition (DD) preconditioning. DD methods take advantage of the partition of the mesh by first imposing artificial boundary conditions on the subdomain boundaries and hence solving each subdomain independently for the interior unknowns. In a second step, an 'interface' problem is solved on the boundaries which gives new boundary conditions for the next step of subdomain solution. Adding the results of the second step to the first gives the new conjugate search direction for the CG algorithm.

The parallel run-time for such a preconditioned CG solver is determined by two factors: the maximum time needed by any of the subdomain solutions and the number of iterations of the global CG search. Whilst an algorithm such as the multigrid method, when used as the solver on the subdomains, is relatively robust against subdomain shape, the number of global iterations are heavily influenced by the subdomain ARs, [25]. Essentially, the subdomains can be viewed as elements of the interface problem, [5], and, just as with the normal finite element method, where the condition of the matrix system is determined by the AR of elements, the condition of the global preconditioning matrix is dependent on the AR of subdomains.

In seeking to optimise the partition on the basis of subdomain AR, rather than cut-weight, it is first necessary to determine a suitable cost function to minimise. In fact it turns out that a particularly appropriate function is given by

$$\Gamma = \sum_p \frac{\partial S_p}{(\Omega S_p)^{\frac{d-1}{d}}}$$

where ∂S_p is the surface area (or perimeter length in 2D) of subdomain S_p, ΩS_p is its volume (area in 2D) and d ($= 2$ or 3) is the dimension of the mesh.

We use the dual graph representation of the mesh (§1.2.2) and so, since every graph vertex represents a mesh element, as in Figure 1(b), each edge corresponds to an element face. However, to use this cost functions in a graph-partitioning context, it is necessary to add some additional qualities to the graph. In particular, vertices must store the volume and total surface area of their corresponding element. More importantly, we also weight the edges of the graph with the size (length in 2D, area in 3D) of the face to which they correspond. This leads to a particularly elegant formulation, since minimising the cost function Γ is equivalent to minimising the cut-edge weights, and so the partitioning algorithms require relatively little modification.

The method is fully described in [32] and is very successful at minimising subdomain AR. The paper also demonstrates that, dependent on the mesh, partitions with good subdomain aspect ratios can vary greatly from those with a low edge-cut.

To *fully* validate the method, it would be interesting to measure the correlation between the definition of aspect ratio used here and convergence in the solver and verify that it does indeed provide the benefits for DD preconditioners that other researchers suggest, e.g. [5, 25]. It would also be interesting to extend the ideas to investigate the 'shaping' of subdomains to reflect anisotropic behaviour.

5.2 Heterogeneous communication networks

As described in §1.2, the usual practice in graph-partitioning is to approximate the communications cost by the cut-weight and then attempt to minimise this quantity. However, many, if not most, parallel machines are based on networks in which the communications cost (both latency and bandwidth) is not uniform across the interprocessor network and in this case the cut-weight is certainly an inadequate measure. For instance, a cut edge between two processors which are 'neighbouring' in some sense will contribute far less to the overall cost than an edge between two processors which are 'far apart'. This is particularly true for symmetric multiprocessor (SMP) clusters – systems of multiprocessor compute nodes with very fast intra-node communications but relatively slow inter-node networks – and for meta-computers – multiple supercomputers combined together, in extreme cases over inter-continental networks (e.g. [7] discusses a meta-computer consisting of two Cray T3Es, one in Stuttgart, the other in Pittsburgh, USA).

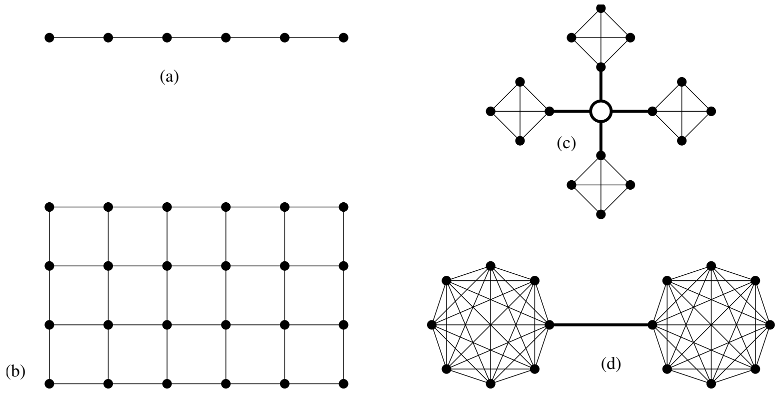
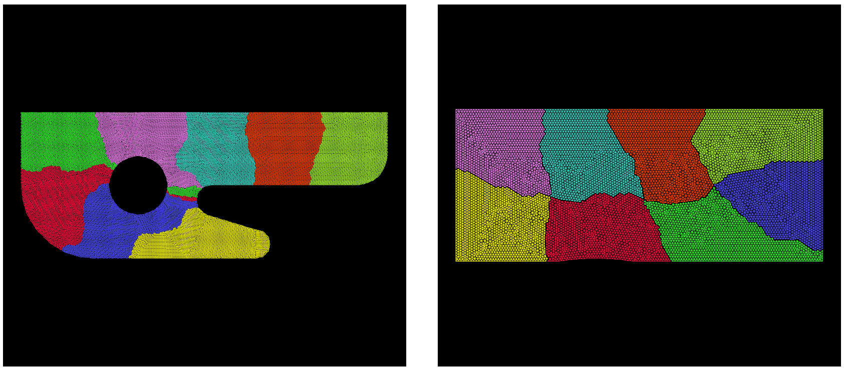


Figure 9: Example processor graphs: (a) 1D array, (b) 2D array, (c) SMP cluster and (d) meta-computer.

Figure 9 shows some typical processor graphs which represent machine interconnection networks. For example, Figure 9(a) is a 1d array, a configuration which may not actually occur in practice but which could be useful for machines with very high communication latencies, since, if the graph can be successfully mapped onto this topology, each subdomain will have at most 2 neighbours. Figure 9(b) is a 2D array (realised in the past by the Intel Paragon and in 3D by the Cray T3D). More recently however, machines have appeared which have a hierarchical network and, for example, Figure 9(c) shows an SMP cluster of 4 compute nodes (each of 4 processors) with all inter-node communications passing through a hub. Finally, Figure 9(d) illustrates a meta-computer.

In [31] Walshaw and Cross describe modifications to JOSTLE which address the mapping problem – a partitioning variant in which the cost of a cut-edge, (v_1, v_2) say, between vertices $v_1 \in S_p$ and $v_2 \in S_q$ is conflated with some network weighting factor representing the relative cost of communication between processors p and q. In fact the coarsening algorithm can be left unchanged and the cost function is first taken into account when the P vertices of the coarsest graph are assigned to the P processors. The modified cost function is subsequently refined on each of the multilevel graphs in succession by relatively simple changes to the gain and preference functions (§2.3.1).

A successful mapping is then one in which subdomains are constructed such that adjacent subdomains generally lie on adjacent processors. The power of the process to compute such a mapping stems from the global properties of the multilevel algorithm. Edges which cross expensive links are penalised heavily within the cost function and so the vertices at either end of such an edge tend to migrate to nearby processors and create a sort of buffer zone. Because this occurs first on the coarser levels of the graph, where each vertex represents many vertices in the original graph, the buffer zone which may start off only one vertex wide, can actually represent reasonably broad regions. In this way the partition is given a good global quality on the coarse graphs which is refined on the finer graphs. Figure 10 shows example graphs mapped onto 1D and 2D arrays, both with 8 processors.



(a) mapping onto a 1D array

(b) mapping onto a 2D array

Figure 10: Example graphs mapped onto two different processor configurations

The technique is described in full in [31] and shows some impressive results. The model of the communications network can be supplied by the user at run-time and the technique is very generic since the mapping algorithm can apply to any architecture, simply by changing the network cost matrix. Indeed a machine could even be instrumented at run-time.

Finally, although the mapping algorithms have not been tested in parallel, there is nothing inherently serial about any of the modifications to the multilevel partitioner and in principle they could be applied to the parallel version described in §4.

5.3 Multiphase partitioning

Multiphase partitioning is a further variant, developed to aid a class of parallel mesh-based solvers characterised by multiple distinct computational subphases, each of which must be balanced separately. Typically multiphase partitioning problems arise from multiphysics or multiphase modelling (e.g. [16]) where different parts of the computational domain exhibit different physical behaviour and/or material properties. They can also arise in contact-impact modelling (see below).

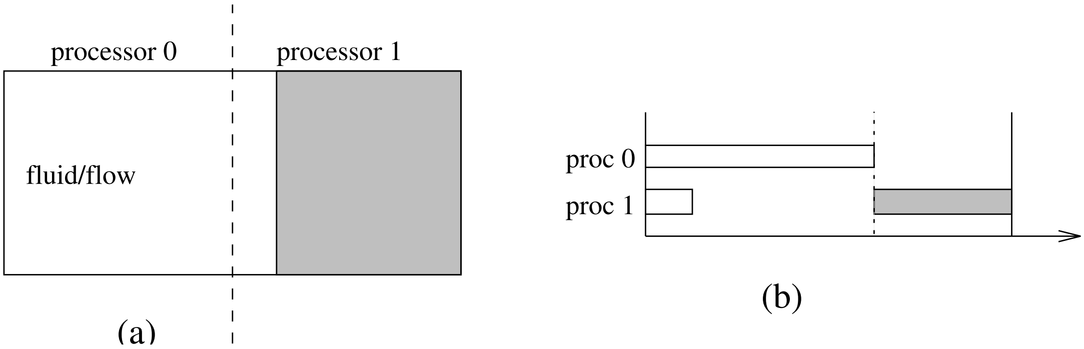


Figure 11: An example of a multiphysics problem showing (a) the partition of the mesh; and (b) the solution time-line.

Consider the example shown in Figure 11(a) with a partition for 2 processors indicated by dotted line. This partition might normally be considered of good quality, but for a solution algorithm which solves for flow and stress separately, it is completely unsuitable because processor 1 is not able to start the solid/stress calculation until the fluid/flow part has terminated. As a time-line of the parallel solver, Figure 11(b) shows, during the fluid/flow phase of the calculation, processor 1 has relatively little work to do and during the solid/stress phase processor 0 has no work at all.

In [34], Walshaw *et al.* address the multiphase partitioning problem by using the standard multilevel partitioner as a ‘black box’ solver, partitioning the problem phase by phase, but calculating the partition of each phase based on that of the previous phases. The strategy is described in detail in [34] and shown to be successful in producing high quality, *balanced* partitions where a standard mesh-partitioning techniques simply fail (as they take no account of the different phases). The paper provides results for three different classes of multiphase problem:

Distinct two-phase problems. In this context, distinct means 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.

Entity-based two-phase problems. 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 [17] and we simulate it taking a set of meshes and assigning the elements to phase 1 and the nodes to phase 2. In such problems the two phases are not well separated with a small interface as above, but highly integrated and very interconnected.

Contact-impact problems. One of the particular areas of interest driving the development of multiphase partitioning algorithms has been the use of contact-impact algorithms (e.g. used for simulating crashes in the automotive industry). 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.

Some additional examples, showing the parallel multiphase version of JOSTLE in action for contact-impact problems, can be found in [2].

5.4 Evolutionary search

In [22, 23] Soper and Walshaw report on two variants of an evolutionary search algorithm, with JOSTLE at its core, which has is able to find partitions considerably better than those found by any of the public-domain graph-partitioning packages. Although this evolutionary technique is not a possible substitute for such packages – the very long run times preclude such a possibility for the typical applications in which graph-partitioning is used – it is of interest to find the best possible partitions for benchmarking purposes. Indeed, for certain applications such as circuit partitioning, where the quality of the partition is paramount, the computational resources required may be completely justified by the very high quality partitions that the technique is able to find.

Evolutionary (or genetic) algorithms produce new search points by one of two operations: crossover, which combines information from two or more randomly selected individuals in the current generation, and mutation which modifies a single, randomly selected, individual. The construction of successful crossover and mutation operators is problem specific and often complex, especially where individuals are subject to constraints (as are the partitions) so that information from different individuals cannot be arbitrarily combined or modified. Further, the information needs to be effectively exploited so that new individuals result that are fitter than the current best individuals with sufficient probability even when the current generation is already very good.

In the first variant, [23], a genetic algorithm is constructed with a crossover that modifies the graph using edge weights to record where the parent partitions had cut-edges. JOSTLE is applied to the new graph, effectively as a local optimisation procedure, and the weighting, or biasing, means that cut-edges of the parents are more likely to be cut again. The mutation operator has a bias which exploits the local translational invariance possessed by many graphs of interest.

More recent work, [22], uses similar operators but further exploits the properties of the graphs being partitioned. Effectively both crossover and mutations act on subdomains (or the set of cut edges containing a subdomain) and the major difference is that JOSTLE needs only to be applied to a fraction – almost always less than half – of the graph to be partitioned. Much more information is transferred into the offspring from the parent(s) and the optimisation algorithm is more effectively focussed on one part of the problem at a time.

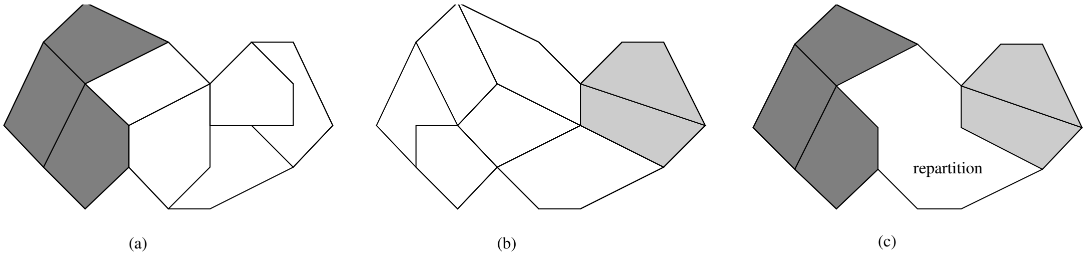


Figure 12: An illustration of the crossover operator

In particular, crossover selects sets of complete subdomains from two individuals, and combines them in the child by partitioning the remainder of the graph. Figure 12 illustrates this, with Figures 12(a) & 12(b) showing two parent partitions which have been selected for crossover. Sets of adjacent subdomains which do not intersect are retained (shown shaded) and the remainder of the graph – the unshaded part of Figure 12(c) – is repartitioned.

Meanwhile, the mutation operator selects a set of subdomains that constitute a cycle in the subdomain graph. The subgraph defined by this cycle is then repartitioned so as to exploit local translational symmetry; new partition boundaries are sought close to existing boundaries where they should have similar and so sometimes less cut edges.

For both papers, the crossover and mutation operators require that certain edges of the graph are made more likely to be cut. This is achieved by adding biases to the edge weights. Mutations are implemented by making existing cut edges and their neighbours much less costly, and crossover by making the cut edges of both parents slightly less costly. Different biases are constructed for every operation (even when the parents are identical) via small randomised additions.

Results are given in [22, 23] and typically provide partitions that have 20% lower cut-weight than those found by a single run of a multilevel partitioner. Of course the comparison is unfair as the evolutionary approach can soak up CPU cycles and for larger graphs has run-times measured in days rather than fractions of a second. However, the techniques provide good benchmark solutions and the resultant partitions still dominate the graph-partition archive[1] over 5 years on.

Finally note that, although this work uses a serial version of the multilevel algorithm, in principle, the same strategy could be used to enable a parallel version of the code by employing the parallel version of JOSTLE. Alternatively, there are many ways to parallelise the evolutionary search algorithm, such as using a processor farm and distributing each partition calculation to an idle processor. However neither strategy has been tested.

6 Summary

We have given an overview of JOSTLE, the parallel multilevel graph-partitioning software, and discussed how it has contributed to research in the partitioning field. In particular, it has helped to enable a variety of applications, most notably parallel mesh-based computational mechanics simulations, and demonstrated that partitioning in parallel is a practical possibility. The algorithms have been extended and modified in a variety of ways and have provided results for a number of partitioning problem variants. Finally, it has helped to investigate and explore the multilevel paradigm for combinatorial optimisation problems.

References

[1] 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.

[2] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic load balancing of finite element applications with the DRAMA library. *Appl. Math. Modelling*, 25(2):83–98, 2000.

[3] 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.

[4] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comput. & Structures*, 28(5):579–602, 1988.

[5] C. Farhat, N. Maman, and G. Brown. Mesh Partitioning for Implicit Computations via Domain Decomposition. *Internat. J. Numer. Methods Engrg.*, 38:989–1000, 1995.

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

[6] 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.

[7] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. In *Proc. Euro PVM/MPI '98, Liverpool*, 1998.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[9] B. Hendrickson. Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? In A. Ferreira and J. Rolim, editors, *Proc. Irregular '98: Parallel Algorithms for Irregularly Structured Problems*, volume 1457 of *LNCS*, pages 218–225. Springer, Berlin, 1998.

[10] 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.

[11] 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.

[12] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In M. T. Heath *et al.*, editor, *Proc. 8th SIAM Conf. Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1997. (CD-ROM).

[13] 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.

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

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

[16] 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.

[17] K. McManus, C. Walshaw, S. P. Johnson, and M. Cross. Partition Alignment in Three Dimensional Unstructured Mesh Multi-Physics Modelling. In C. A. Lin *et al.*, editor, *Parallel Computational Fluid Dynamics: Development & Applications of Parallel Technology*, pages 459–466. Elsevier, Amsterdam, 1999. (Proc. Parallel CFD'98, Taiwan, 1998).

[18] J. Savage and M. Wloka. Parallelism in Graph Partitioning. *J. Parallel Distrib. Comput.*, 13:257–272, 1991.

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

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

[21] H. D. Simon and S.-H. Teng. How Good is Recursive Bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[22] A. J. Soper and C. Walshaw. A Generational Scheme for Partitioning Graphs. In L. Spector *et al.*, editor, *Proc. Genetic & Evolutionary Comput. Conf. (GECCO-2001)*, pages 607–614. Morgan Kaufmann, San Francisco, 2001.

[23] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[24] M. Toulouse, K. Thulasiraman, and F. Glover. Multi-level Cooperative Search: A New Paradigm for Combinatorial Optimization and an Application to Graph Partitioning. In P. Amestoy *et al.*, editor, *Proc. Euro-Par '99 Parallel Processing*, volume 1685 of *LNCS*, pages 533–542. Springer, Berlin, 1999.

[25] D. Vanderstraeten, C. Farhat, P. S. Chen, R. Keunings, and O. Zone. A Retrofit Based Methodology for the Fast Generation and Optimization of Large-Scale Mesh Partitions: Beyond the Minimum Interface Size Criterion. *Comput. Methods Appl. Mech. Engrg.*, 133:25–45, 1996.

[26] C. Walshaw. An Exploration of Multilevel Combinatorial Optimisation. In J. Cong and J. Shinnerl, editors, *Multilevel Optimization in VLSICAD*, pages 71–123. Kluwer, Boston, 2003. (Invited chapter).

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

[28] C. Walshaw. Variable partition inertia: graph repartitioning and load-balancing for adaptive meshes. In S. Chandra M. Parashar and X. Li, editors, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*. Wiley, New York, 2006. (Invited chapter).

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

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

[31] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. *Future Generation Comput. Syst.*, 17(5):601–623, 2001.

[32] 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.

[33] 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.

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

[35] C. Walshaw and M. G. Everett. Multilevel Landscapes in Combinatorial Optimisation. Tech. Rep. 02/IM/93, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, April 2002.