RALPAR-LIB - A Multilevel Partitioning Library

R F Fowler and C.Greenough

March 1998

Abstract

This report describes the multilevel partitioning methods which have been implemented in the software package Ralpar. Such methods address the problem of finding an efficient partitioning of a mesh for parallel processing in a reasonable time. A graph representation of the connectivity of a mesh is condensed through a number of levels to give a smaller problem that can be partitioned quickly. Kernighan and Lin refinement is used to improve the partition on intermediate levels. A library interface to these partitioning routines is described, the Ralpar multilevel library (RPMLL). Some examples are given to illustrate the performance of this implmentation of multilevel methods.

A copy of this report can be found at the Department's web site (*http://www.dci.clrc.ac.uk/*) under page *Group.asp?DCICSEMSW* or anonymous ftp server *www.inf.rl.ac.uk* under the directory *pub/mathsoft/publications*

Mathematical Software Group
Department for Computation and Information
Rutherford Appleton Laboratory
Chilton, DIDCOT
Oxfordshire OX11 0QX

Contents

1 Introduction

The Ralpar software tool is a mesh partitioning program which implements a range of methods for splitting unstructured meshes for parallel processing purposes. The documentation for the original version of this program [14] describes a number of partitioning methods that have been implemented within it. These methods vary in their computational complexity and the quality of the partitions that they can produce but the *best* methods, such as spectral bisection tend to be expensive.

Several authors have recently investigated the use of multilevel techniques as a way of achieving high quality mesh partitioning at lower cost than conventional methods [4, **?**]. These methods seek to generate a hierarchy of simpler "meshes" from the original mesh so that a low cost partitioning of the smallest mesh can be made and mapped back in some way to solve the full problem. It has been shown that these can give substantial savings in time over the spectral bisection method, while retaining the high quality results of this method.

Hence developments have been made to Ralpar to include multilevel partitioning methods. The details of the multilevel methods are described in this document. We also present some measurements on real meshes to illustrate how the multilevel methods can be used and to help in the selection of parameters to optimise performance.

Ralpar was originally developed as a stand alone program for mesh partitioning which would read a mesh file and output a corresponding partition file. This is useful for the investigation of methods and their performance, but is not convenient when using Ralpar as part of a real parallel analysis system. Hence all the multilevel methods have been developed to allow them to be called as library functions. The library interface, known as RPMLL (RalPar MultiLevel Library), is also be described in this document.

The structure of the report is as follows. In Section 2 we briefly review the importance of mesh partitioning for parallel processing. We then look in detail at the way in which multilevel methods can be used and how they have been implemented in Ralpar. Some examples of the results obtained with the RPMLL method within Ralpar are then given in Section 5 and some conclusions drawn in the final section. The Appendix gives details of the interface routines that can be used to access the RPMLL and suggestions for parameter values to use.

2 Mesh partitioning for parallel processing

2.1 The importance of mesh partitioning

Many physical problems can be most efficiently solved using a mesh of points to approximate the continuous functions in space and time. Such applications include computational fluid dynamics, electromagnetic analysis and semiconductor device simulation. In all these cases, fully three dimensional and time dependent analysis of complex structures is a highly computationally intensive task. Such problems demand the use of high performance machines and efficient algorithms. The first requirement means use of parallel computers. The latter requirement often leads to the use of unstructured meshes which can be more easily adapted to accurately represent the solution with fewer points than finite difference methods.

There are many ways in which a mesh based calculation can be mapped onto a parallel machine and it may be the case that the most efficient serial algorithm, often selected on the basis of the lowest flop count, is not the optimal one for a parallel machine. One example of this is a solution processes that depends on solving linear systems using iterative methods. The conjugate gradient algorithm is very efficient if used with incomplete Cholesky preconditioning in the serial case. However incomplete Cholesky preconditioning is highly serial and can lead to poor performance on parallel machines where it is often either modified or replaced with another preconditioner. Even a poor preconditioner such as diagonal scaling may prove more effective on a parallel machine as it avoids the serial bottleneck.

The importance of finding a good partitioning of the data for parallel computations is well known, see for example [9] and [10]. A simple illustration can be seen by looking at the example of sparse matrix vector product operation $\mathbf{y} = A\mathbf{x}$. This operation is at the heart of many iterative linear solvers. Typically each value x_i will corresponding to an unknown at grid node i, and the non-zeros in A reflect how the mesh connects node i to its neighbouring nodes. If both the matrix A and the vector $\mathbf{x}$ have been partitioned between two processors, then the matrix vector product can be written as

$$\begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} \tag{1}$$

where the subscripts 1 and 2 now refer to the separate partitions. Thus $\mathbf{y}_1 = A_{11}\mathbf{x}_1 + A_{12}\mathbf{x}_2$ and this part of the product will be stored on processor 1. The first term on the right is all stored on processor 1. The second term, $A_{12}\mathbf{x_2}$, requires fetching the product results from processor 2. If the partitioning can make the block matrices A_{12} and A_{21} as sparse as possible then there will be less data to transfer and hence less time wasted in interprocessor communication.

In addition to the requirement to minimise the amount of communication, there is also the need to ensure load balance so that each processor has, as close as possible, the same amount of work to do [1] . For Equation 1, assigning half the nodes to each processor, so that the vectors are of the same length should give load balance as long as the average sparsity of A_{11} and A_{22} is the same. For many mesh based computations this assumption is quite good, but there are cases where computation per entity is highly non-uniform. For these situations it is necessary to weight each point i according to its computational cost w_i. Thus if the computational power of each processor is C_k and there are n nodes and p processors, load balance requires that the ideal total weight of nodes for processor j, W_j, is just:

$$W_j = C_j \sum_{i=1}^{n} w_i / \sum_{k=1}^{p} C_k. \tag{2}$$

As long as the computational work per node and the the power of each processor are well known (and this is not always the case) it is not difficult to ensure that a partitioning will give good load balance. The hard part of the problem is in finding a partitioning that also minimises the communication costs.

2.2 Graph based representation of partitioning

The way data is partitioned will depend on the details of the computation being performed on the mesh. In some cases nodal variables are used while in others element based variables or even mesh edge based variables may be needed. Sometimes a mixture of these will be used.

[1]In the case of a heterogeneous system each processor should have an amount of work proportional to its computational power for load balance.

Hence the first decision to be made is which entity is be partitioned. The standard version of Ralpar assumes that elements will be partitioned and that nodes and mesh edges will be shared between processors if the elements either side are assigned to different processors. This is convenient when the software spends most time on calculation of element contributions.

The geometric partitioning algorithms discussed in [14] all work on the physical position of the mesh entities. This means that distortions of the mesh which do not change the element connectivity can lead to different partition results. This is not a desirable property and is reflected in the fact that the quality of geometric mesh partitions is highly variable.

Non-geometric partitioning methods make use of the mesh connectivity described by the element topology. Because the connectivity of nodes differs from that of elements it is often convenient to work in terms of the undirected graph which describes the connectivity of each entity. Figure 1 illustrates the graph corresponding to a simple 2D mesh. In this example each graph vertex represents an element in the real mesh. Edges exist in the graph only between vertices representing neighbouring elements.

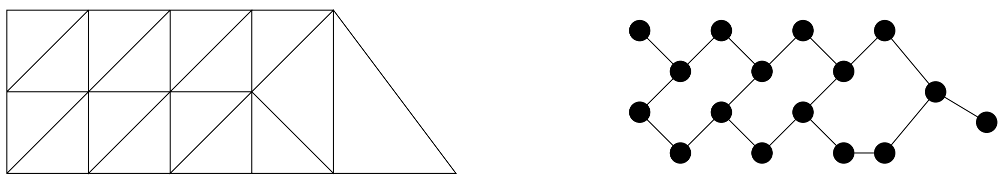

Figure 1: *A simple mesh is shown on the left with the corresponding graph of element connectivity on the right. This is the edge communication graph, where elements are connected if they share a common mesh edge. In this case, with linear triangular elements, the graph of nodal connectivity would look similar to the original mesh.*

A similar graph could just as easily be built with the vertices representing nodes or mesh edges. For the case of the element connectivity graph in Figure 1 we have assumed that 2D elements are connected if they have a common mesh edge and that these connections all have the same weight. Ralpar allow three different types of element connectivity graph to be generated from the mesh. These are:

1. The *edge communication graph.* Connections exist between elements that have a common mesh edge in 2D or a common face in 3D. All connections have unity weight.

2. The *true communication* graph. Connections exist between any two elements that share one or more common nodes. Again all connections have unity weight.

3. The *weighted communication* graph. This also has connections between any two elements with one or more common nodes. However the weight of the connection is equal to the number of common nodes.

Note that the connections between vertices in the graph are usually called edges of the graph and should not be confused with edges of the mesh.

The choice of which graph is more appropriate depends on the communication requirements of the parallel algorithm to be used. For example, if a parallel computation only requires the exchange of data between elements with a common face, then the first type of graph is appropriate. Each edge in the graph should represent a need for communication between processors if the vertices at either end are on different processors.

Once a graph representation of the mesh has been obtained it is then necessary to consider how it can be partitioned in light of the requirements of load balance and minimisation of communication costs. The load balance problem can be addressed as in Equation 2. As long as the computational cost of each entity is known, along with the power of the processors to be used, the total weight of graph vertices to be assigned to each partition can be determined.

The problem of minimisation of communication costs is usual taken as been equivalent to minimisation of the number of *cut edges*. A cut edge is a connection in the graph between two vertices that have been assigned to different processors. In the case of a weighted graph the aim is to minimise the weighted sum of cut edges. For the two processor partitioning in Equation 1, each cut edge corresponds to an extra entry in the off-diagonal blocks A_{12} and A_{21}.

Though we have concentrated on mesh based calculations and the graphs that arise from these, similar problems can occur in many other areas. Electronic VLSI design (for the placement of components with minimum interconnect) and sparse linear algebra problems are two such examples where graph partitioning is also important. In some of these cases there are no coordinates associated with the graph vertices. This precludes the use of any geometric partitioning schemes.

2.3 Graph partitioning methods

The main graph partitioning techniques used in Ralpar are discussed in detail in [14]. Most methods are based on finding an ordering of the vertices within the graph. Once this is obtained, the graph is then split into two (or more) parts by picking the required number of vertices from the ordered list. The multilevel partitioning library makes use of the following graph partitioning methods:

Graph bisection: From any vertex in a graph we can label the set of neighbouring vertices as level 1 and work out from there labelling their neighbours as level 2 (excluding any that have already been labelled), etc., until all vertices in the graph are labelled by the number of steps they are away from the original vertex. If this operation is repeated a number of times, selecting as the starting vertex each time the last one labelled in the previous pass, this tends to a numbering between the two vertices with the greatest separation, known as the maximum diameter of the graph [8]. The basic step of this graph partitioning method is then to divide the vertices of the graph according to their level numbers.

Dual graph bisection: A slight variation of the graph bisection method is available within RPMLL. In this method we identify two vertices, one at each end of the graph diameter, using the level numbering approach as above. Instead of gathering vertices from just one end of the graph, we gather them from both ends in an alternating fashion. Vertices from the first end are labelled from 1 upwards while those from the other end are labelled downwards from N_v downwards, where N_v is the number of vertices in the graph. In this case we are using sequence numbers rather than level numbers. This method generally requires use of a refinement technique with it to give reasonable results.

Bandwidth and profile methods: Malone [12] used standard bandwidth reduction algorithms to obtain orderings to split up a mesh. Ralpar includes the Gibbs-King and Gibbs-Poole-Stockmeyer bandwidth and profile reduction algorithms using the an implementation from netlib/ACM [6]. The Malone implementation used bandwidth minimisation for the nodes and then transfered this numbering to elements. Within RPMLL profile minimisation is used to directly renumber the graph vertices and then partition using this ordering.

Greedy methods: The Greedy method described by Farhat [9] is normally implemented in terms of nodes and elements of the mesh. In RPMLL it is applied directly to the graph. The starting point for gathering vertices is taken as the vertex of lowest order (fewest connections). Vertices are labelled in order out from the starting point in a similar way to the graph bisection method. This ordering is then used to split the graph. Since there will often be more than one possible choice of starting vertex for each partition, it is also possible to use the Glutton variant of the greedy method [14].

Spectral bisection: This method makes use of the second smallest eigenvector of the associated Laplacian matrix to split the graph. The method, due to Pothen *el al* [2], is computationally expensive. Within Ralpar we use the LASO package [7] to determine the appropriate eigenvector.

Kernighan and Lin: The algorithm of Kernighan and Lin [11] does not explicitly generate a partition but can be used to try and improve an existing one, while retaining load balance. Each vertex of the graph is inspected and the change in cut edges of swapping it from its current partition to the the other partition is evaluated. A single pass of the KL method proceeds to swap vertices in alternating directions to maintain balance. The vertex to be swapped is always that which gives the best decrease in the number of cut edges (or the smallest increase). Once a vertex has been swapped it is not moved again in the current KL pass. After each swap, cost values for all neighbour vertices must be updated. This must be done carefully to avoid the cost of each pass being quadratic in the number of vertices.

Random partitioning: A random split of the vertices at each step can be made. This is only useful if used with a refinement method such the KL algorithm. It has the advantage of looking at starting configurations which are not generated by the conventional methods.

2.4 Measures of partition quality

The main measure of partition quality used by the graph based methods described above is the total number of cut edges resulting from the partitioning. This is also the objective function used in the KL refinement process.

Another measure that may be of use in some cases is the maximum number of partitions about any single partition. With the Ralpar interface to RPMLL the maximum and average of the neighbours about a partition is reported. The start up time to initiate a communication operation can sometimes be very significant and may favour methods, such as those of Malone, which tend to reduce the number of neighbouring partitions.

For graphs describing element connectivity we can also determine the number of interface nodes that are generated by a partitioning. An interface node is one that lies on the boundary between two or more partitions. The calculation of the number of interface nodes requires access to the element topology information as well as the actual graph partitioning. This measure can be useful in comparing the partitions generated using different graphs of element connectivity (edge communication, true communication, etc.).

Appendix A gives details of the routines `RPPARQAL` and `RPNODCST` which can be used to determine the above values for a given partitioning of a graph.

3 Multilevel methods

3.1 Previous work on multilevel techniques

Multilevel methods have been used by several authors for speeding up the process of graph partitioning. The basic idea is to merge neighbouring vertices of the graph together in some fashion so as to produce a smaller, more manageable graph. The merging process can be repeated several times until a graph that is sufficiently small has been obtained. This process is illustrated in Figure 2.

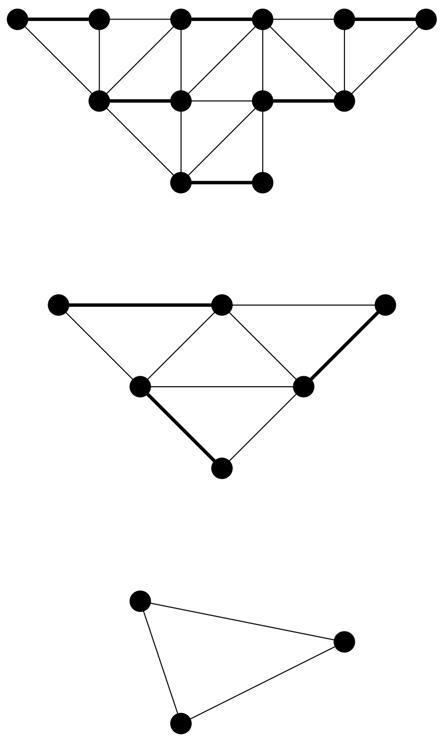

Figure 2: *An illustration of how a graph can be condensed by merging vertices. The top graph has 12 vertices. By merging those vertex pairs which have been connected with a bold line we get the second graph with just 6 vertices. Another level of condensation leaves just 3 vertices.*

There are at least two main approaches to the use of multilevel techniques in graph partitioning.

- Barnard and Simon [1] and others, e.g. [3], have used multilevel methods as a means to efficiently calculate the eigenvalues to be used in spectral bisection of the full graph.

This is usually referred to as multilevel spectral bisection (MSB). It has been shown that obtaining the second smallest eigenvalue on the simplest graph and using this to approximate the corresponding eigenvectors for the higher level graphs can be very effective in reducing the CPU time to find the eigenvector of the full graph.

- Multilevel partitioning schemes on the other hand try to partition the lowest level graph directly. Having got a partition at this level they then propagate it back through all the intermediate graphs up to the original graph. Usually some refinement is made at each intermediate level to improve the quality of the partition. Methods of this type have been used by Hendrickson and Leland [4] and Karypis and Kumar [5].

The implementation within Ralpar is based on the latter approach and offers a number of options as to the partitioning method to be used on the coarse graph. In addition the user can control the way in which the graph is reduced and the refinement to be used on intermediate levels.

3.2 Graph condensation methods

In the example shown in Figure 2 a simple 2D graph was reduced from 12 vertices to 3 by a process of merging neighbouring vertices two at a time. Clearly there are many different ways in which pairs these could be chosen. It is also possible to merge more than two vertices at a time to give a more rapid condensation of the graph with fewer levels. It is likely that a more rapid condensation with fewer intermediate levels will be more efficient in terms of memory and computational speed. On the negative side it is possible that larger clusters will have more uneven interfaces giving lower partition quality and requiring more refinement work on the higher level graphs.

The current version of Ralpar currently supports two methods of vertex clustering. These are as follows:

Greedy clustering: A scan is made through all vertices in the graph. If a vertex has not yet been clustered it is merged with all its neighbouring vertices which have not yet been clustered. All these vertices are then marked so that they will not be clustered any further in this level.

Heaviest edge clustering: Again all vertices are scanned through. When an unclustered vertex is found, all its neighbouring unclustered vertices are examined and it is merged with the one which has the highest edge weight connecting it to the first vertex.

Figure 3 demonstrates how these two methods work on a simple graph.

It is important to track the edge weights as a graph is condensed, even if the original graph is unweighted. The weight of an edge in the condensed graph is just the sum of the weights of edges joining the two clusters in the higher level graph. This values should be proportional to the amount of communication required if the two clusters are assigned to separate processors.

The weight of vertices in the condensed graph is similarly given be the sum of weights of the merged vertices. This must be available to ensure load balance in partitioning the condensed graph. Vertex and edge weights are shown in Figure 3 assuming that the initial graph has unit weights for both of these quantities.

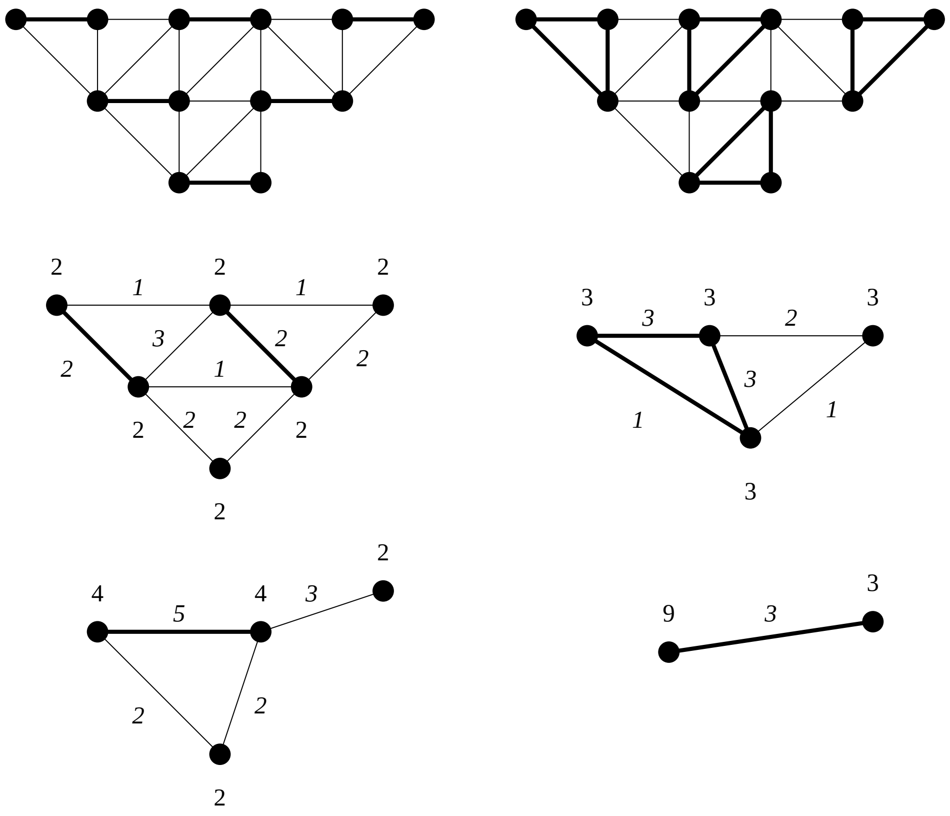

Figure 3: *Clustering vertices: the three levels on the left have been clustered using the heaviest edge method. The first level is assumed to have no edge weighting, so vertices are merged starting from the top right with the first free neighbour. For the weighted graphs, vertex weights are given in a Roman font and edge weights in italic. The three graphs on the right show greedy clustering. Each vertex is merged with all free neighbours. This leads to a more rapid reduction in graph size, as can be seen here.*

3.3 Partitioning condensed graphs

All the graph partitioning algorithms previously used within Ralpar have assumed that graph edges are unweighted. Some of these methods can be used without modification to partition weighted graphs. This is partly because there is no obvious way to to adapt them to use the additional information. The set of partitioning methods that have been implemented in RPMLL can be divided into two with respect to the effect of edge weighting. Those that have not been adapted to take account of edge weighting are:

Graph bisection: The level numbering scheme which is at the heart of the graph bisection algorithm does not take the weight of an edge into account. The same applies to the dual graph variation of this method.

Bandwidth and profile methods: Again the renumbering methods only make use of the fact that a link exists, not on the actual weight of the edge.

Those methods that have been altered in some way to use the edge weighting information are:

Greedy methods: The Greedy method is similar to graph bisection in the gathering stage, and the weight of an edge has no influence on this part of the method. In the selection of the starting vertex though, it is necessary to allow for the edge weight to find a vertex of minimum connectivity. In the Glutton variation of the standard greedy method several starting vertices are tried and the one with lowest cut edge cost is selected. The evaluation of cut edge cost must take into account the edge weights.

Spectral bisection: In the unweighted case the Laplacian matrix L has off-diagonal entries $L_{ij} = -1$ whenever an edge exists between vertices i and j. For the weighted graph case, this value is replaced by the negative of the actual edge weight. The diagonal terms, L_{kk}, are then given by the sum of weights of all edges meeting at vertex k. The rest of the spectral bisection algorithm remains unchanged.

Kernighan and Lin: The basic steps of the KL smoothing method are unchanged, but the weight of edges must be taken into account when computing the cost changes that result from any vertex partition swap. While the actual change to the cost computation is trivial, this does have some repercussions in the efficient implementation of this algorithm. This is due to the fact that it is necessary to keep a sorted list of the cost of each possible vertex swap. To avoid doing an expensive full sort of this list on every swap, a list of swaps at all possible cost levels is kept. This table can become large and sparse for heavily weighted edges, making the method less efficient.

3.4 Uncoarsening and partition refinement

Once a partition has be made on the lowest graph level it is a simple matter to map it back through all the intermediate graph levels to the original graph. At each level of graph condensation it is necessary to retain an array recording the mapping of vertices in the fine graph to the coarse one. This gives a rapid partitioning of the full graph from that on the smallest graph. However the load balance of the partition will be governed by the best that could be obtained on the coarsest graph and the partition boundaries will be forced to follow the cluster boundaries.

To improve the partition quality and the load balance, KL refinement of the partitions can be used. This is straightforward when a simple partition of a graph into two parts is considered as the refinement can be done as the solution is mapped back from one graph to the next. When more than two partitions are required the cost of refinement becomes more significant because the first bisection has to be transfered back up to the original graph. Refinement of this graph means that the original condensation of the graph is now no longer valid as some clusters will straddle partition boundaries. Hence the condensation process has to be repeated for each bisection operation.

4 Software Implementation

4.1 Language

All of Ralpar and the multilevel library has been developed in standard Fortran 77. The exception is at the start of the Ralpar software where the workspace is allocated with a call to a C subroutine. Though C does offer some advantages in memory allocation and wider range of data structures, it was felt that Fortran was more a more appropriate choice as it allows easy access to standard sparse eigenvalue software.

4.2 Graph representation

Since the RPMLL can be called from other software packages it is necessary to know the details of the graph representation that has been used. Though the library does include routines to generate graphs from the users topology information, these arrays must be declared by the calling program. Communication graphs are stored as a pair of integer arrays:

- `PG(NV)` This array is a set of pointers into the graph array `G`. The length must be $N_v + 1$, where N_v is the number of vertices in the graph.
- `G(1:IG,1:JG)` This array stores the neighbours of each vertex and must be addressed via `PG`. For vertex `I` the neighbours will be stored in locations `G(1,J)` where `J` runs from `PG(I)` to `PG(I+1)-1`. Note that all neighbours of each vertex are stored and this format does not take account of symmetry. The first dimension, `IG`, may be either 1 or 2. The former case implies an unweighted graphs where all edges have unity weight, while in the latter case `G(2,J)` gives the weight of edge `J`. Only integer edge weights are allowed.

In addition to the above data describing the graph, it is necessary to provide an array of vertex weights `GW(NV)`. These are required even if all weights are unity. The array must be integer and all vertex weights are strictly positive.

4.3 Graph generation

The graph is built from the mesh topology information. To construct the graph for element partitioning the list of elements about each node is first formed. This just requires knowing the set of nodes that make up each element of the mesh. To make the software easier to integrate into other programs, the element node information is accessed via a function call.

This minimises the dependence of this operation on the format used to store the topology data - it is just necessary to modify this routine to extract the required information.

Once the list of elements about nodes has been obtained the graph can be built using it. As mentioned previously there are three options available for the type of graph to generate. The edge graph assumes that elements are connected if they have a common face. The true communication graph assumes a connection if there is a least one node in common, while the weighted graph weights each link according to the number of common nodes.

A similar graph can be built for the nodes instead of the elements. In this only one option is supported, where a node is connected by unweighted links to all the other nodes in each element it appears in. The user is free to generate graphs in other ways if they are more appropriate to the problem under consideration.

4.4 Generation of multilevel graphs

Assuming that a graph G_0 has been generated from the mesh, then we wish to build a hierarchy of smaller graphs from it, $G_1, G_2 \ldots, G_N$. The procedure used to do this is as shown in Figure 4.

```
**Procedure** gen_mlgraphs ( $G_0$ , MaxLev , MinSize )
$i = 0$
**do** while $i <$ MaxLev and Size($G_i$) $>$ MinSize
        v = find_vertices_to_merge( $G_i$ )
        $G_{i+1}$ = generate_subgraph( $G_i$, $v$ )
        $i = i + 1$
**enddo**
**end**

**Procedure** find_vertices_to_merge ( $G$ )
v = **0**
$c = 0$
**do** $i = 1, N_v$
        **if** ( $v_i \equiv 0$ ) **then**
                $v_i$ = c
                **for all** $j$ where $G(i,j) \neq 0$ and $v_j \equiv 0$
                        $v_j = c$
                $c = c + 1$
        **endif**
**enddo**
**end**
```

Figure 4: *Outline of the method to generate a set of graphs from the original graph G_0. The vertex clustering operation here is of the greedy form where all available vertices are merged at each step. N_v is the number of vertices in the current graph and* v *is a vector giving the mapping of vertices on a fine graph to those on the next coarser level.*

The amount of space taken to build the set of condensed graphs depends on the graph itself and the clustering method used. As each subgraph should be at least half the size of the previous level, the total should not exceed that of the original graph. A single workspace array is used as input to the RPMLL and all working arrays are allocated from this.

Note that all subgraphs will include edge weighting and vertex weights. In addition the clustering data at each level will have to be stored with the graph.

4.5 Multilevel methods

To partition a graph into n parts with a multilevel method there are two approaches available within RPMLL. The first is faster but does not allow refinement on intermediate levels because we alway work on the lowest level graph and only map the final partition back to the original graph. The second method involves mapping the partition back to the original graph after each bisection. This allows refinement of intermediate levels, but is more expensive as the condensation process has to be repeated each time.

The control flow in the two methods is outlined in Figure 5.

The pseudo code in Figure 5 we have used G to denote a graph and P to indicate a partitioning vector which maps vertices to partitions. Several complex operations are abbreviated by single statements such as *partition(* G_n *,* N *)* to denote a call to a standard partitioning method to split the vertices of the graph into N parts, *KL_smooth(* P^s_k*,*G^s_k*)* for an application of the Kernighan and Lin algorithm to improve the given bisection on the current graph. In general the quality of partitions produced without refinement of the intermediate levels limits the usefulness of the first method.

The above outline of the method with KL refinement is structured for the case of recursive bisection. This is normally the most efficient method to split up a mesh. This method is not limited of cases where the number of partitions is a power of 2, since weighting can be used to get the required number of partitions. The multilevel library also supports splitting graphs using *recursive sectioning*, where one complete partition at a time is split away from the main body of the graph. This can also be combined with KL refinement on the intermediate graphs. Another option is to use *linear sectioning*, where the the partitioning method is called just once to provide an ordering of all vertices in the graph. This is then used to split the graph into the required number of partitions. Linear sectioning can not currently be combined with KL refinement, hence it is only supported for the first type of partitioning in Figure 5.

4.6 The library interface

In addition to the implementation of multilevel methods within Ralpar, the partitioning methods can also be accessed via a library interface. This allows the methods to be included into other software packages more easily. There are two main partitioning routines plus a number of support routines which can be used to help in building the initial graph and assessing the quality of the generated partition. All the interfaces to these routines are described in Appendix A.

A simple test program, *rplibtest.f* is included in the distribution version of Ralpar which illustrates the use of the multilevel library interface. The program can be used to generate a partitioning of a graph described in a simple ASCII formatted file, but is also useful as a guide to include RPMLL in other programs.

```
**Procedure** ml_partition_1 ( $n$ , $G_0$ )
$G_1, \ldots G_n$ = gen_mlgraphs( $G_0$ , MaxLev , MinSize )
$P_n$ = partition( $G_n$, $N$ )
**do** $i = n$ to 1 step $-1$
    $P_{i-1}$ = map_partition( $P_i$ )
**end**

**Procedure** ml_partition_2 ( $n$ , $G_0$ )
set initial partition $P_0$ to all in partition 1
**do** $i = 1$ , $log_2 n$
    **do** $j = 1$ , $2^{i-1}$
        $G_0^s$ = generate_subgraph( $G_0$, $P_0$, $j$ )
        $G_1^s, \ldots G_n^s$ = gen_mlgraphs( $G_0^s$ , MaxLev , MinSize )
        $P_n^s$ = partition( $G_n^s$, 2 )
        **do** $k = n$ to 1 step $-1$
            $P_{k-1}^s$ = map_partition( $P_k^s$ )
            KL_smooth( $P_{k-1}^s$, $G_{k-1}^s$ )
        **enddo**
        update $P_0$ from $P_0^s$
    **enddo**
**enddo**
**end**
```

Figure 5: *Pseudo code outlining the two different implementations of multilevel methods in Ralpar. The first method condenses the graph once, performs the normal partitioning on that graph and maps the result back to the top level. In the second method, the bisection partitioning method has been modified so that a set of condensed graph is generated for each partition every time it is to be bisected.*

5 Examples

To illustrate the use of the multilevel partitioning methods within the Ralpar framework we now present some example results on 2D and 3D meshes. A more detailed comparison of the multilevel method with other techniques will be made in [16].

5.1 Simple 2D structured mesh

A simple mesh is shown in Figure 6 which is actual a structured grid composed of quadrilateral elements, but is useful to illustrate the multilevel methods.

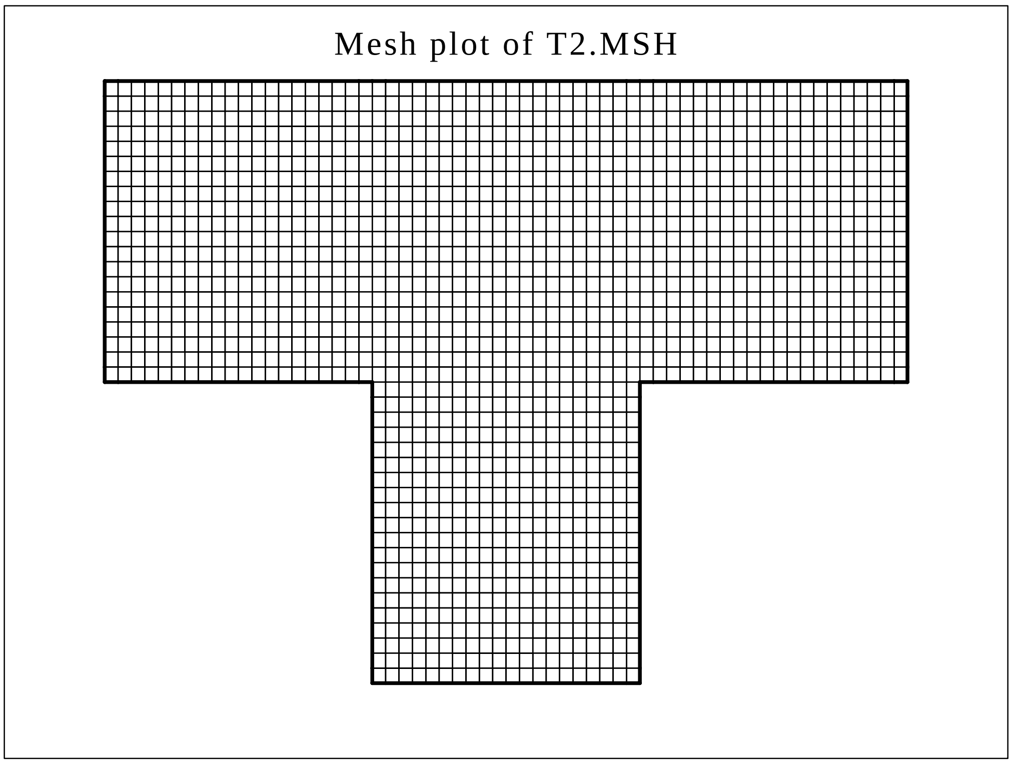

Figure 6: *A simple structured mesh of 1701 nodes and 1600 quadrilateral elements.*

One of the main advantages of multilevel techniques is in saving CPU time in expensive methods such as recursive spectral bisection. In Figure 7 we show three results of partitioning the test mesh into 7 parts of equal size. The only difference between the three cases is in the number of levels of graph condensation that has been used. The first case, with `MAXLVL=0` corresponds to no condensation and is essentially a non-multilevel result. In the other two cases we have used 1 and 2 levels of graph condensation with the greedy clustering method. Within Ralpar the command to generate such a partition with 2 levels of condensation would be:

```
MLPART 7 SPEC KLEF=FULL GTYPE=E MAXLVL=2
```

Full details of the commands and their parameters are given in the Ralpar User Manual [15].

It can be seen that there is a wide variation in the actual partitions that are obtained when graph condensation is used, at least in this particular case. Some of these differences

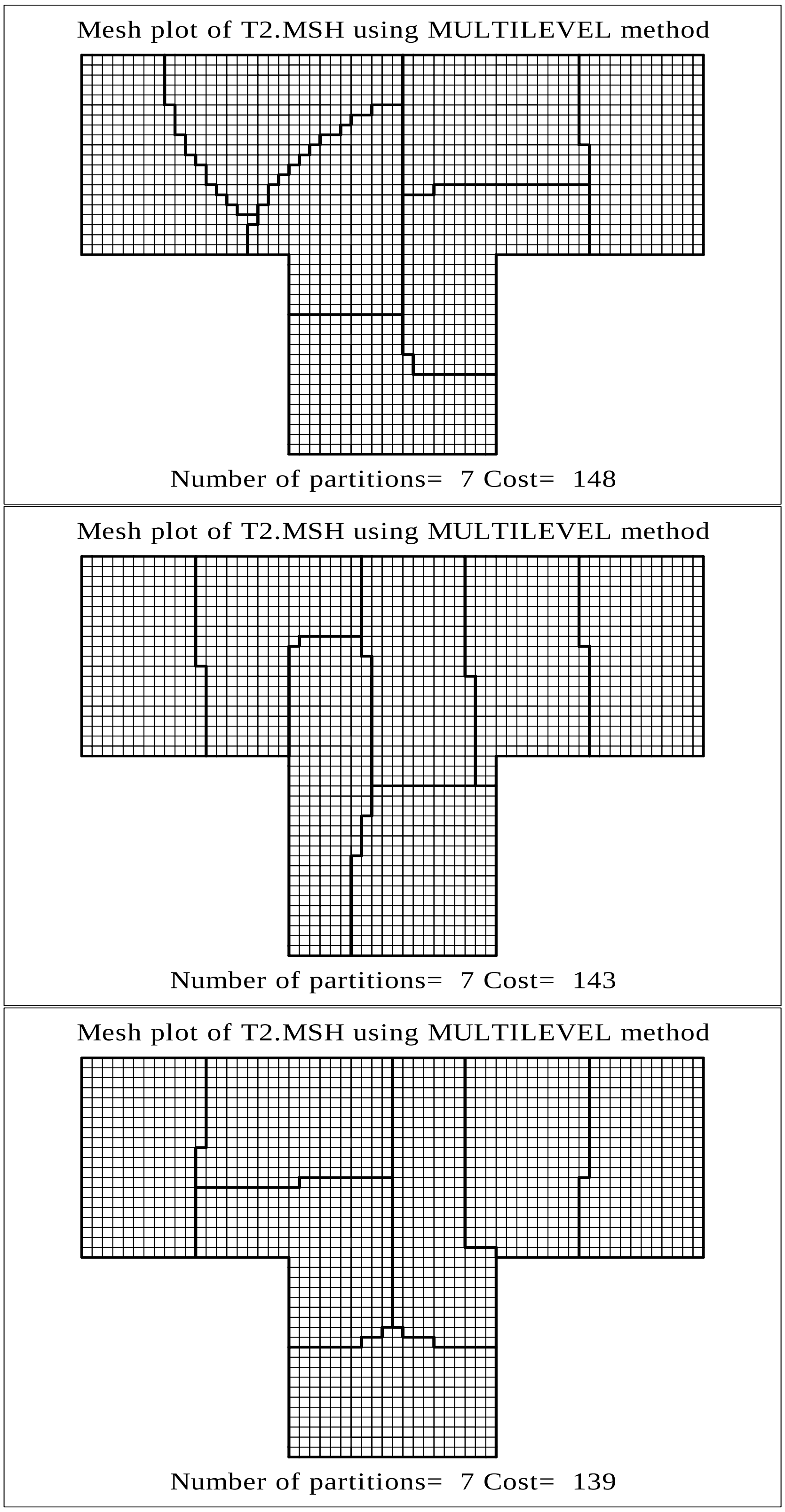
Mesh plot of T2.MSH using MULTILEVEL method
Number of partitions= 7 Cost= 148
Mesh plot of T2.MSH using MULTILEVEL method
Number of partitions= 7 Cost= 143
Mesh plot of T2.MSH using MULTILEVEL method
Number of partitions= 7 Cost= 139

Figure 7: *The three figures correspond to partitioning the graph into 7 parts using recursive spectral bisection with KL refinement on all intermediate levels. The top figure is with no condensation, the middle with one level and the third with 2 levels.*

are due to KL refinement which is performed on all graph levels, though even without any refinement there are still significant differences. Table 1 shows the partition quality measures (mesh interface nodes and graph cut edges) as a function of the number of levels used. The CPU times were obtained on a Sparc 10 processor using full compiler optimisation.

Max. Level	Int. Nodes	Cut Edges	CPU (secs)
0	148	147	4.00
1	143	140	1.45
2	139	136	0.65
3	138	135	0.43
4	143	139	0.36

Table 1: *Partition quality and CPU time as a function of the maximum number of levels used in the multilevel method. Seven partitions are generated using spectral bisection and KL refinement.*

In this simple test case the measures of partition quality get slightly better as more graph levels are used, up to 3 levels. It is not always the case that the multilevel version produces better results but allowing the KL refinement method to work on a range of size scales can sometimes be beneficial. In terms of CPU time it can be seen that the multilevel version is an order of magnitude faster using 3 or 4 levels of condensation. Again this result only applies to this particular test case, but is representative of the sort of speed up possible through the use of multilevel methods. Some further speed up may be obtained by use of the parameter `KLLIM` which limits the length of KL refinement passes.

As well as using spectral bisection to provide the initial partition on the lowest level graph, other methods can be used. As an example, Table 2 shows the results of using graph bisection as the splitting method. These results are again for generation of 7 partitions.

Max. Level	Int. Nodes	Cut Edges	CPU (secs)
0	138	134	0.26
1	146	142	0.20
2	148	143	0.29
3	135	132	0.28
4	145	142	0.29

Table 2: *Partition quality and CPU time used as a function of the maximum number of levels used in the multilevel method. Seven partitions are generated using graph bisection and KL refinement.*

In this case there is little gain to be seen in CPU time from using multilevel methods. This is to be expected since graph bisection on its own is a very fast technique. Though the quality results compare favourably with those from use of spectral bisection, this is not always the case.

5.2 2D unstructured triangular mesh

A unstructured mesh of the space around an airfoil is shown in Figure 8. This mesh is due to Hammond and can be found via the WWW. The mesh contains 4720 nodes and 9000 elements.

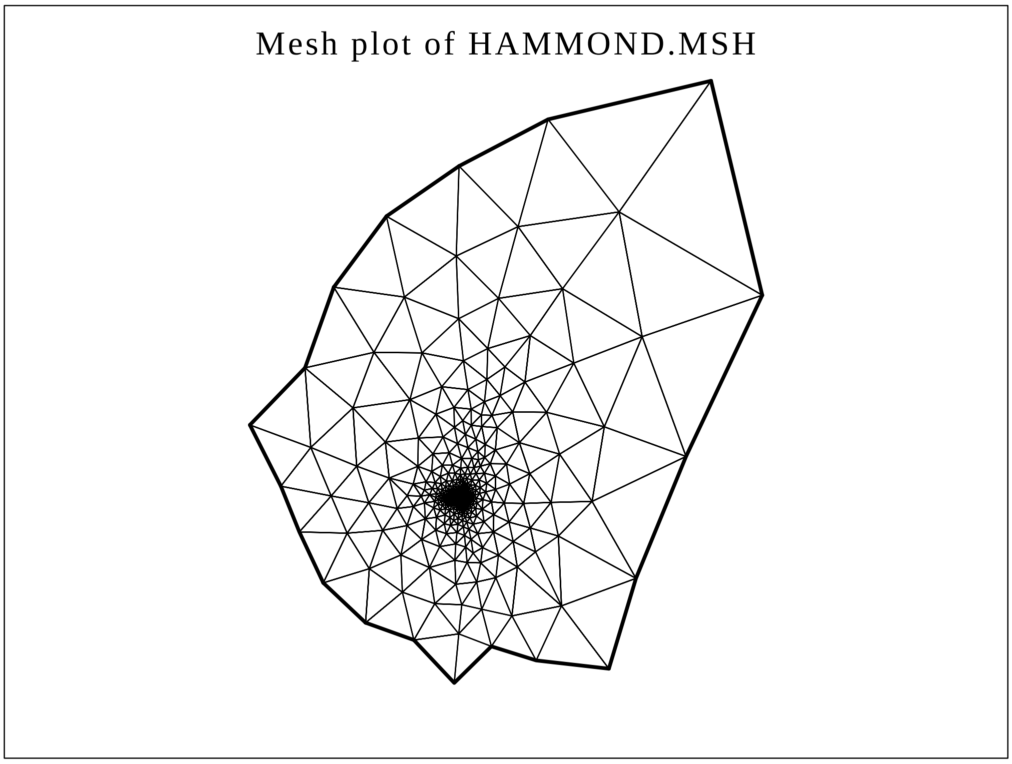

Figure 8: *The Hammond mesh. The aerofoil at the centre of the mesh is not visible on this scale.*

In Table 3 we compare some measures of the partition quality and CPU time for splitting the Hammond mesh with multilevel methods. Partitioning has been made of the elements of the mesh, rather than the nodes. Again we have used the edge communication graph for these tests with greedy clustering and KL refinement on all levels.

It can be seen that the multilevel methods offer substantial savings in CPU time as the number of levels is increased. Again there is often a slight improvement in the partition quality as the the number of levels increases. The use of the parameter `KLLIM=200` is seen to have a slight effect in reducing the CPU time used, though sometimes giving more cut edges. The speed up by use of multilevel methods is of the order of 20 in these cases.

5.3 3D unstructured hexahedral mesh

As a final example we present some results of partitioning a 3D unstructured hexahedral mesh of the airspace within a cylinder head. The mesh is shown in Figure 9 and contains 26573 nodes and 23446 elements.

A set of multilevel results are summarised in Table 4 for the case of partitioning the graph into 8 parts. Again a speed up of an order of magnitude is seen through the use of the multilevel techniques. In a 3D mesh the greedy clusters will grow faster than in 2D meshes. This may explain the fact that there is less improvement above two levels of condensation than seen

Partitions	Max. Level	Int. Nodes	Cut Edges	CPU (secs)
2	0	55	54	16.79
	1	55	54	7.53
	2	54	53	2.30
	3	51	50	1.28
	4	51	50	0.78
	$4^\dagger$	51	50	0.59
8	0	229	226	52.90
	1	218	215	15.16
	2	204	201	7.44
	3	206	203	3.25
	4	204	202	2.23
	$4^\dagger$	204	202	2.02
64	0	928	939	69.10
	1	908	922	28.00
	2	874	889	12.20
	3	892	907	5.67
	4	898	912	4.42
	$4^\dagger$	899	914	4.07

Table 3: *Results for the Hammond mesh. Partition quality and CPU time as a function of the maximum number of levels used. The symbol † indicates that the KLLIM parameter was set to 200.*

previously. The larger mesh also shows a more significant reduction in CPU time through the use of the `KLMIM` parameter.

Max. Level	Int. Nodes	Cut Edges	CPU (secs)
0	2071	1915	53.20
1	2025	1866	31.30
2	1973	1807	10.67
3	2085	1937	10.37
4	2132	1934	10.31
$4^\dagger$	2088	1909	5.21

Table 4: *Partition quality and CPU time results for the multilevel method on the cylinder head mesh. Results are for partitioning into 8 parts.* † *indicates the use of the KLLIM parameter.*

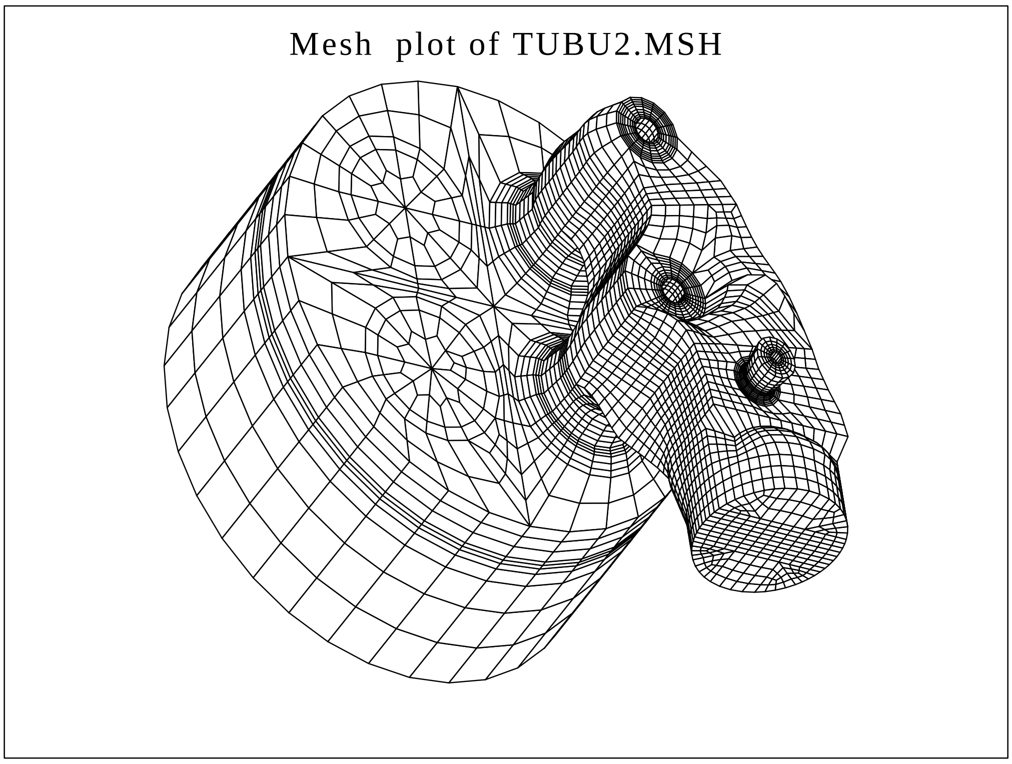

Figure 9: *A 3D mesh of a cylinder head.*

6 Conclusions

A multilevel partitioning library has been developed for use in graph partitioning. This library offers a number of options to control the clustering method and how many levels are used. A range of partitioning methods, including spectral bisection, may be used to split the lowest level graph and Kernighan and Lin type refinement can be used on all graph levels to improve the partition quality.

Results presented here show that the multilevel methods can easily give an order of magnitude speed gain compared to spectral bisection and KL refinement on a single level. In addition we have seen that partition quality from the multilevel methods is as good, and often slightly better than, that of the single level methods.

A more detailed comparison of the performance of multilevel methods with other partitioning techniques will be made in [16].

References

[1] ST Barnard and H Simon: "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems", *Proc. of the sixth SIAM conf. on Parallel Processing for Scientific Computing*, 711-718, (1993).

[2] A Pothen, HD Simon and KP Liou: "Partitioning sparse matrices with eigenvectors of graphs", *SIAM J. of Matrix Analysis and Applications*, **11**, No. 3, p430-452, (1990).

[3] B Hendrickson and R Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations", *Technical Report SAND92-1460*, Sandia National Laboratories (1992).

[4] B Hendrickson and R Leland, "A multilevel algorithm for partitioning graphs", *Technical Report SAND93-1301*, Sandia National Laboratories (1993).

[5] G karypis and V Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", *University of Minnesota Technical Report: 95-035* (1996).

[6] Algorithm 582, Collected Algorithms from ACM, ACM Trans. Mathematical Software, **8**, No. 2, p. 190 (1982).

[7] DS Scott, "LASO - Block Lanczos Software for Symmetric Eigenvalue Problems", ORNL/CSD-48 (November 1979). Code from netlib.

[8] YF Hu and R Blake, "Numerical experiences with partitioning unstructured grids", Daresbury Laboratory Report, DL/SCI/P865T, March 1993.

[9] C Farhat, W Wilson and G Powell: "Solution of Finite Element Systems on Concurrent Processing Computers" *Engineering with Computers*, **2**, 157–165, (1987).

[10] YF Hu and RJ Blake: "Numerical Experiences with Partitioning Unstructured Meshes", Daresbury Laboratory Report, DL/SCI/P865T, March 1993.

[11] B Kernighan and S Lin: "An efficient heuristic procedure for partitioning graphs", Bell System Technical Journal, 29 (1970), pp. 291-307.

[12] JG Malone: "Automatic Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessor Computer", Comp. Meth. in Applied Mechanical Eng., **70**, 27–58 (1988).

[13] Algorithm 582, Collected algorithms from ACM, ACM-Trans. Math. Software, Vol. 8, No. 2, p. 190, June 1982.

[14] C Greenough and RF Fowler: "Partitioning Methods for Unstructured Finite Element Meshes", Rutherford Appleton Laboratory Report, RAL-94-092.

[15] RF Fowler and C Greenough: "RALPAR - RAL Mesh Partitioning Program, Version 2.0", Rutherford Appleton Laboratory Report, RAL-98-025, (1998).

[16] C Greenough and RF Fowler: "A Review of Partitioning Methods for Unstructured Finite Element Meshes", Rutherford Appleton Laboratory Report, (To be published).

A The library interface

In addition to the implementation of multilevel methods in Ralpar, the partition methods can also be accessed via a library interface. This allows the methods to be included into other software packages more easily.

This appendix lists the top level subroutines that are provided within the RPMLL. Three types of routines are provided for:

- Graph generation
- Graph partitioning
- Partition quality assessment

The details of the parameters for each subroutine are described along with suggested values where appropriate.

A.1	RPPARMLL	Partition graph, refinement of lowest graph only
A.2	RPPARMLF	Partition graph, refinement of all intermediate graphs
A.3	RPGENGCL	Generate graph of cell connectivity
A.4	RPGENGND	Generate graph of node connectivity
A.5	RPEDCUT	Evaluate cut edges in a partitioning
A.6	RPLDBAL	Evaluate load balance of a partitioning
A.7	RPNODCST	Evaluate interface nodes in cell partitioning
A.8	RPPARQAL	Evaluate quality and connectivity of a partitioning

A.1 Graph partitioning routine RPPARMLL

Syntax

```
   SUBROUTINE RPPARMLL(PG,IPG,G,IG,JG,GW,NV,NP,PW,PMTH,WORK,IWORK,
  +                    COPT,MAXLVL,MINSIZ,KLREF,KLLIM,GP,INFO,IERR)
```

Description

This subroutine partitions a graph using multilevel methods *without* KL refinement on the intermediate levels.

Parameters

PG(IPG)	Integer array of pointers into graph G, where IPG is equal to $N_v + 1$.
G(IG,JG)	Integer array describing the graph. Vertices linked to I are given in G(1,PG(I)) to G(1,PG(I+1)-1). If the graph is weighted, IG=2, then G(2,I) is the edge weight. Unweighted graphs have IG=1. JG is the number of edges in the graph. Note that all links must be included even though the graph is undirected and hence symmetric.
GW(NV)	Integer array giving the weights for each vertex.
NV	Integer, the number of vertices in graph, N_v.
NP	Number of partitions required.
PW(NP)	Real*4 array giving the partition weights.
PMTH	Integer giving the partition method. This encodes both the splitting method and the way in which it is applied. The available methods are: 1. Profile (Malone) renumbering method. 2. Spectral method. 3. Graph method. 4. Random partitioning. 5. Dual graph method. 6. greedy method. 7. Glutton method. The way in which a method is used is controlled by adding a constant to the above method numbers. For recursive bisection, add 100, for recursive sectioning, splitting one partition off at a time, add 200. If no constant is added, then linear sectioning is used, where the separator is evaluated only once and used to split the graph into N parts.

`WORK(IWORK)`	Integer workspace array. The exact size required by this array is dependent on the method used and the number of levels of condensation. Twice the size of the graph array would be a reasonable initial guess.
`COPT`	Integer denoting how graph vertices should be condensed. `COPT=1` implies use the greedy clustering method. `COPT=2` implies use the heaviest edge clustering method.
`MAXLVL`	Integer giving the maximum number of levels to use. Typical values may be in the range 0 to 5.
`MINSIZ`	Integer giving the minimum size of graph that should be generated. Typical values may be 10 to 1000 vertices.
`KLREF`	Logical flag to control KL refinement. If this is true, then KL refinement will be made on the partitions generated at the lowest graph level. This must be false if a linear section method is used.
`KLLIM`	Integer value to control KL refinement. If this is zero then the full KL method is applied. If it is set to a value > 0 then each KL pass can terminate early if the current cost gain is `KLLIM` below the best one seen so far. Typical values are in the range 100 to 5000.
`INFO`	Integer value to control the output of information. Zero gives the lowest amount of output. For more details on the operation of the software, values in the range 1 to 3 may be used.
`GP(NV)`	Integer array giving the partitioning of the graph. For each vertex this gives the partition it has been assigned to.
`IERR`	Integer error flag. This should always be checked on return. A zero value indicates success.

A.2 Graph partitioning routine RPPARMLF

Syntax

```
   SUBROUTINE RPPARMLF(PG,IPG,G,IG,JG,GW,NV,NP,PW,PMTH,WORK,IWORK,
  +                    COPT,MAXLVL,MINSIZ,KLREF,KLLIM,GP,INFO,IERR)
```

Description

This subroutine partitions a graph using multilevel methods *with* KL refinement possible on all the graph levels. Note that most parameters are identical to those of subroutine `RPPARMLL`, so the description given above are brief.

Parameters

`PG(IPG)`	Integer array of pointers into graph `G`, where `IPG` is equal to $N_v + 1$. (In).
`G(IG,JG)`	Integer array describing the graph. (In).
`GW(NV)`	Integer array giving the weights for each vertex. (In).
`NV`	Integer, the number of vertices in graph, N_v. (In).
`NP`	Number of partitions required. Options are the same as for `RPPARMLL` except that the linear section method is not supported. (In).
`PW(NP)`	Real*4 array giving the partition weights. (In).
`PMTH`	Integer giving the partition method. (In).
`WORK(IWORK)`	Integer workspace array. The exact size required by this array is dependent on the method used and the number of levels of condensation. Twice the size of the graph array would be a reasonable initial guess. (Workspace).
`COPT`	Integer denoting how graph vertices should be condensed. (In).
`MAXLVL`	Integer giving the maximum number of levels to use. (In).
`MINSIZ`	Integer giving the minimum size of graph that should be generated. (In).
`KLREF`	Logical flag to control KL refinement. If this is true, then KL refinement will be made on all graph levels. (In).
`KLLIM`	Integer value to control KL refinement. (In).
`INFO`	Integer value to control the output of information. (In).
`GP(NV)`	Integer array giving the partitioning of the graph. For each vertex this gives the partition it has been assigned to. (Out).
`IERR`	Integer error flag. This should always be checked on return. A zero value indicates success. (Out).

A.3 Graph generation routine RPGENGCL

Syntax

```
   SUBROUTINE RPGENGCL(NCELLS,NNODES,GTYPE,DIMEN,WORK,IWORK,PPG,IPG,
  +                    PG,IG,JG,IPT,IERR)
```

Description

This subroutine generates a graph of cell based connectivity in the format required by RPMLL.

Parameters

`NCELLS`	Integer, number of cells or elements in the mesh. (In).
`NNODES`	Integer, number of nodes in the mesh (In).
`GTYPE`	Integer, for type of graph to generate. Values are 1 for edges graph, 2 for true communication graph and 3 for weighted graph. (In).
`DIMEN`	Integer giving dimensionality of mesh, 2 or 3. (In).
`WORK(IWORK)`	Integer array of work space, but which will contain the graph on exit. The size of `IWORK` depends on the type of mesh, but should be at least twice the expected size of the graph array. (Out).
`PPG`	Integer, pointer into array work to the start of the array of pointers to the graph. (Out).
`IPG`	Integer, the length of the array of pointers to the graph. (Out).
`PG`	Integer giving the start location of the graph data in `WORK`. (Out).
`IG, JG`	Integers giving the dimensions of the graph array stored in `WORK`. (Out).
`IPT`	Integer giving the first unused location in the array `WORK`. (Out).
`IERR`	Integer error flag. This should always be checked on return. A zero value indicates success. (Out).

User provided subroutine

The user has to provide a subroutine which will return the nodes which form a given cell or element. This routine must be called `RPUCLINF` and take the arguments:

```
   SUBROUTINE RPUCLINF(CELL,NODES,INODES,NODCL)
```

`CELL`	Integer giving the number of the cell for which information is requested (1 to `NCELLS`). (In).
`NODES(INODES)`	Integer array that will contain the nodes forming cell `CELL`. (Out).
`INODES`	The maximum number of nodes in a cell. This is set internally to 32 but can be increased if necessary. (In).
`NODCL`	Integer giving the number of nodes in the requested cell.(Out).

Keeping strictly to the Fortran 77 standard requires that the cell topology data is stored in a common block for this routine to access the required data.

A.4 Graph generation routine RPGENGND

Syntax

```
   SUBROUTINE RPGENGND(NCELLS,NNODES,GTYPE,DIMEN,WORK,IWORK,PPG,IPG,
  +                      PG,IG,JG,IPT,IERR)
```

Description

This subroutine generates a graph of the nodal connectivity. The set of parameters are the same as that required for `RPGENGCL`.Parameter `GTYPE` must have the value 2 at present and this generates a graph where a node is connected to all other nodes that appear in a common cell. This routine also requires the existence of the user supplied subroutine`RPUCLINF` to give the nodes in each cell.

A.5 Partition assessment routine RPEDCUT

Syntax

```
SUBROUTINE RPEDGCUT(GP,PG,IPG,G,IG,JG,NV,CEDGES,IERR)
```

Description

This subroutine evaluates the number of cut edges in a graph given the graph and the partition vector.

Parameters

`GP(NV)`	Integer array giving the partition that each vertex has been assigned to. (In).
`PG(IPG)`	Integer array of pointers into graph `G`, where `IPG` is equal to $N_v + 1$. (In).
`G(IG,JG)`	Integer array describing the graph. (In).
`NV`	Integer giving the number of vertices in the graph. (In).
`CEDGES`	Integer giving the number of cut edges in the graph. (Out).
`IERR`	Integer error flag. Zero for success. (Out).

A.6 Partition assessment routine RPLDBAL

Syntax

```
SUBROUTINE RPLDBAL(NV,GP,GW,PW,NP,RBAL,RBMAX,IERR)
```

Description

This subroutine reports the quality of the load balance achieved by the partition vector `GP` in relation to the requested partition weights `PW`. Normally we expect good load balance but this may not be the case when multilevel methods are used without refinement on the intermediate levels.

Parameters

`NV`	Integer giving the number of vertices in the graph. (In).
`GP(NV)`	Integer array giving the partition that each vertex has been assigned to. (In).
`GW(NV)`	Integer array giving the weights for each vertex. (In).
`PW(NP)`	Real*4 array giving the requested partition weights. (In).
`NP`	Integer giving the number of partitions that have been generated. (IN).
`RBAL(NP)`	Real*4 array giving ratio of partition weight to that requested by `GW`. Ideally all these should be unity. (Out).
`RBMAX`	Real*4 value giving the largest value of `RBAL`. (Out).
`IERR`	Integer error flag. Zero for success. (Out).

A.7 Partition assessment routine RPNODCST

Syntax

```
   SUBROUTINE RPNODCST(NCELLS,NNODES,NPNOD,NP,GP,TOTAL,COST,NTYPE,
  +                    CLNOD,ICLNOD)
```

Description

This subroutine is specifically designed for use with partitioning graphs that represent cell or element connectivity. For such a partitioning it calculates the number of mesh nodes that lie on internal partition boundaries.

Parameters

`NCELLS`	Integer, number of cells or elements in the mesh. (In).
`NNODES`	Integer, number of nodes in the mesh (In).
`NPNOD`	Integer, number of periodic nodes pairs in mesh. (In).
`NP`	Integer giving the number of partitions that have been generated. (IN).
`GP(NCELLS)`	Integer array giving the partition that each cell has been assigned to. (In).
`NTYPE(NNODES)`	Integer array, used as workspace. (In).
`CLNOD(ICLNOD)`	Integer array, used as workspace. Length must be at least the maximum number of nodes in a cell. (In).
`TOTAL`	Integer, giving the total number of interface nodes. (Out).
`COST(NP)`	Integer array giving the number of interface nodes associated with each partition. (Out).

User provided subroutines

This routine also requires access to the cell topology information. This requires the existence of `RPUCLINF`, described in the section on `RPGENGCL`. Also required is a subroutine called `RPUPNINF` which returns periodic node pairs for cases where the mesh has a symmetry plan. If no such plane exists then a dummy routine stub can be used as long as `NPNOD` is zero.

A.8 Partition assessment routine RPPARQAL

Syntax

```
   SUBROUTINE RPPARQAL(PG,IPG,G,IG,JG,GP,NV,NP,CEDGES,AVECON,MAXCON,
  +                    AVENEI,MAXNEI,WORK,IWORK,INFO,IERR)
```

Description

This subroutine returns the edge cut cost of a partition, as does RPEDCUT, but also gives information on the average and maximum connectivity of partitions. The number of partitions that a given partition is connected to can be important in estimating the total communication costs.

Parameters

PG(IPG)	Integer array of pointers into graph G, where IPG is equal to $N_v + 1$. (In).
G(IG,JG)	Integer array describing the graph. (In).
GP(NV)	Integer array giving the partition each vertex has been assigned to. (In).
NV	Integer giving the number of vertices in the graph. (In).
NP	Integer giving the number of partitions. (in).
WORK(IWORK)	Integer workspace array, the size of which must be at least 2*(NP**2+NV)+1. (In).
INFO	Integer value controlling amount of output. Should be zero if data is just required via the arguments. (In).
CEDGES	Integer giving the number of cut edges in the graph. (Out).
MAXCON	Integer giving the maximum number of cut edges on any single partition. (Out).
AVECON	Real*4 giving the average number of cut edges on a single partition. (Out).
MAXNEI	Integer giving the maximum number of partitions about any single partition. (Out).
AVENEI	Real*4 giving the average number of partitions about a single partition. (Out).
IERR	Integer error flag. Zero for success. (Out).