AL
atory
ton Laboratory

RAL Report
RAL-94-091

RALPAR - RAL Mesh Partitioning Program Version 1.1

R F Fowler and C Greenough

August 1994

Rutherford Appleton Laboratory Chilton DIDCOT Oxfordshire OX11 0QX

RALPAR - RAL Mesh Partitioning Program
Version 1.1

RF Fowler and C Greenough

May 1994

Abstract

This report describes the first release of the Rutherford Appleton Laboratory Mesh Partitioning Program *ralpar*. *Ralpar* is a software tool to split up unstructured meshes into subdomains for parallel processing on MIMD type architectures. A wide range of basic decomposition methods have been implemented within this tool. This document describes the methods that have been implemented, how to run the tool and the measures of partition quality that are provided.

Mathematical Software Group
Computational Modelling Division
Rutherford Appleton Laboratory
Chilton, Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

This document describes Version 1.1 of the *ralpar* mesh partitioning tool. This tool implements a wide range of basic mesh partitioning algorithms for unstructured finite element type meshes in two and three dimensions. Mesh partitioning for unstructured meshes is important in the area of parallel processing for grid based calculations.

The structure of this document is as follows. Section 2 outlines the requirements for parallel processing of mesh based computations and Section 3 looks at some basic measures of partition quality. We then present an overview of all the algorithms that have been implemented within the tool. Sections 5 looks at how the measures of partition quality can be related to expected communication time on some parallel systems. Much of this work is in an early stage within this release of the software. Finally, Section 6 shows some examples of how the tool can be used.

Appendix A is a command reference Section for *ralpar*, while Appendix B gives a brief introduction to the AVS interface to the partitioner, *avsralpar*. The interface to RALBIC neutral files and memory management control are discussed in Appendices C and D respectively.

2 Parallel processing and mesh decomposition

The use of computing systems with some type of parallel architecture has grown significantly over the past few years. These systems are seen as the path by which sufficient computing power can be provided for accurate three dimensional simulations of complex phenomenon in areas such as fluid dynamics, stress analysis and electromagnetism. Such simulations typically use meshes or grids containing 10^5 to 10^6 nodes and may employ automatic grid refinement during the solution. Unstructured grids are often necessary due to complex geometries and the need to place nodes in an intelligent manner, to minimise the overall error in the solution.

Many of the parallel machines now being used are based on the MIMD form of parallelism where the memory of the machine is distributed over a network of processors. A consequence of this is that the program and its associated data must also be distributed between these processors. In finite volume and finite element methods this leads to the problem of how to distribute large unstructured grids and meshes initially, and how to redistribute them subsequently, if refinement is made. This tool only addresses the problem of the initial partitioning and assumes that a single processor with sufficient memory is available to partition the complete mesh.

When moving a large grid based computation to a distributed memory machine there are two main approaches:

- Direct parallelisation of the existing solution algorithm. In this case data exchanges are made between processors whenever required by the algorithm.
- True Domain Decomposition (DD) methods. In this case separate problems are solved on each of the subdomains independently and then some method is used to "patch" together these solutions, which again involves interprocessor communication. Two subclasses within DD methods are:
    1. Overlapping subdomains. In this case a layer of mesh elements on the interface between two subdomains is assigned to both of them. These allow methods based on the Schwarz alternating procedure [1] to be used.

2. Non-overlapping subdomains. In this case it is necessary to solve a separate global problem for the interface nodes between domains. One approach is to form the Schur complement matrix for the interface.

The current version of *ralpar* only produces non-overlapping mesh decompositions.

Three of the major requirements for efficient parallel processing are:

1. To minimise the sequential part of the computation.
2. To ensure that all processors have the same amount of work to do.
3. To minimise interprocessor communication.

The fact that *ralpar* is a sequential program is not optimal with regard to the first point, but the cost of partitioning is often quite small compared to the actual computations performed on the mesh.

Load balancing is often achievable by assigning the same number of elements to each processor. However in many cases, such as with a mixed mesh containing several different element types, it can be desirable to attach a computational weight w_i to each element. Any partitioning of the mesh should then aim to assign each domain a set of elements with weight W given by

$$W = \frac{\sum_i w_i}{N} \tag{1}$$

The current version of the tool only deals with element partitioning, though there can be cases where a partitioning of the nodes is more appropriate.

Another extension that is becoming important for computations on heterogeneous workstation clusters is that of domain weighting. It may be that one machine is more powerful than the others and hence should have a larger share of the work. This can also be included in the above scheme and is available in the current version of *ralpar*.

Minimisation of interprocessor communication time is very important in distributed grid based calculations. An ideal case would be where the subdomains could be completely decoupled and no exchange of information was required. This is very rarely the case and main goal in mesh partitioning methods is to find a splitting that gives the lowest communication cost. The actual communication time will depend on the characteristics of the parallel machine, the solution algorithm and the nature of the problem being solved.

3 Measuring partition quality

The true measure of interest for any mesh partitioning method is the effect it has on the run time for the computations that are to be performed on a real parallel system. Generally, this effect is too sensitive to the particular combination of algorithm and parallel hardware being used to be practical. Instead simpler models are used which measure the size of the interface between subdomains and related quantities.

Load balancing, by assigning the same amount of work to each processor, is built into all the decomposition schemes discussed here. Hence it is not a good measure to compare decomposition methods with.

As was noted in the previous section, the number of interface nodes that a mesh decomposition produces can effect the size of the interface problem to be solved. This quantity is taken as our main

measure of interface size. For all methods *ralpar* reports the total number of interface nodes that are generated.

Another measure that may be of importance is the number of neighbouring subdomains about each subdomain. This can be important in matrix assembly operations for node based quantities where each processor will have to exchange interface data with all its neighbours. While the total amount of data to be exchanged will be proportional to the number of interface nodes, there is always a certain amount of time involved in initiating a communication with another processor. Under some circumstances this time can be a significant amount of the total communication time. In *ralpar* the user is provided with the average number of neighbouring subdomains for the partition along with the minimum and maximum number of neighbours for any one subdomain.

For the graph based partitioning methods (these are described in the following section) *ralpar* gives the number of *cut edges* generated by the partition. These are links (or edges) in the communication graph that go between two different subdomains. This measure is not identical to the number of interface nodes, but they are roughly proportional to each other. For certain computations, such as matrix assembly for element face based quantities, the amount of communication can be proportional to the number of cut edges.

4 Mesh partitioning techniques

The following set of mesh partitioning techniques are available within *ralpar*:

- Geometric bisection (`geo-bis`)
- Cost Optimised geometric bisection (`costgeo`)
- Greedy algorithm of Farhat (`greedy`)
- Cost optimised version of Greedy algorithm (`glutton`)
- Malone bandwidth minimisation algorithm (`bandwdt`)
- Variation of Malone method with profile minimisation (`profile`)
- Geometric bisection on axes of inertia (`inertia`)
- Recursive version on Inertia method (`r-inertia`)
- Kernighan and Lin method with a Greedy start (`kl-greedy`)
- Kernighan and Lin method with a random start (`kl-rand`)
- Graph bisection method (`graph`)
- Kernighan and Lin with a graph bisection start (`kl-rgb`)

These methods are described in more detail in the following sections.

4.1 Geometric bisection methods

There are a number of methods that are based on the geometric location of elements (or nodes) in space. These methods do not make any direct reference to the connectivity of the mesh. In the case of element partitioning it is usual to use the centroid of each element, determined as the average over the nodal position vectors, as the location.

A simple sort (costing about O($NlogN$) for N elements) can then give ordered lists of vertices along any of the principal axes (x, y or z in three dimensions). A bisection then cuts the mesh in two by first assigning all vertices to domain 1 and then transferring vertices to domain 2 according to the order in the sorted list along one of the axes. We stop when the weight of vertices in the new domain is equal to (or greater than) that left in domain 1. If more than 2 domains are required the bisection process can be applied recursively to subdomains to generate 4, 8, 16,... domains.

In the simplest version of this method (`geo-bis`), the first bisection is made on the x axis, the second level bisections on the y axis and so on. This can give reasonable results on meshes that have similar levels of refinement in all directions. However, it can give poor results for long thin meshes, where a bisection parallel to the longest axis will generate a much larger interface than a bisection perpendicular to it.

An improved version of geometric bisection is to use the cut axis that gives the smallest interface cost each time. To do this we need to be able to measure the interface size in some way. In the cost geometric bisection method (`costgeom`) we select the cut direction that generates fewest new interface nodes each time.

The above types of geometric bisection always use the cartesian coordinate directions x, y, z. However in general there is no reason for the mesh to "align" with these directions and the results will not be invariant to rotation of the mesh. An alternative is to use the principal axes of inertia of the mesh, determined by assuming unit mass at each vertex. The basic `inertia` method that has been implemented just determines the principle axes for the initial mesh and then proceeds to use these in all subsequent bisections, as in the `geo-bis` method.

The `r-inertia` method applies the process recursively, and determines a new set of principle axes for each subdomain that is to be bisected. In addition it evaluates the interface node cost of each of the possible cut directions and selects that with the lowest value.

4.2 Greedy methods

The Greedy algorithm of Farhat [3] just tries to form one partition at a time without regard to how this will affect partitions that are formed later. For each partition the algorithm finds a seed point in the remaining mesh and gathers the required number of elements working out from this point. If we have N elements and p partitions, and assume that $N_p = N/p$ elements are required for each partition, then the algorithm proceeds as follows:

1. Form nodal weighting array of the number of elements about each node in the mesh.
2. do $i = 1$ until p

    2.1 Locate the node with minimum available weight
    2.2 Label all elements attached to this node; they initiate the list of elements of the i^{th} subdomain
    2.3 Recursively add to this list the elements adjacent to those already labeled,

starting always from the first labelled element in the previous step.
2.4 Remove used elements from list and update nodal weights.

This process is repeated until the complete mesh is partitioned.

Some variations of this method are possible. A modified Farhat method has been proposed by Fowler *et al* [4].This is based on the observation that at each cycle the algorithm selects a new minima as its next starting point. It is usually the case that there are a number of nodes which have the minimum weight. In our implementation of the standard Farhat method we just select the first such node that is found. The modified algorithm seeks to try a limited number of the alternative starting points at each cycle of the algorithm. It does this by always selecting the m^{th} minima found as its seed point for each partition, where m is varied between 1 and (by default) 5. We then select the value of m that gives the smallest interface, measured by the number of interface nodes generated. The number of trials has to be limited because the total number of choices increases rapidly with the number of partitions. This is available as the `glutton` option within the `partition` command.

4.3 Graph bisection method

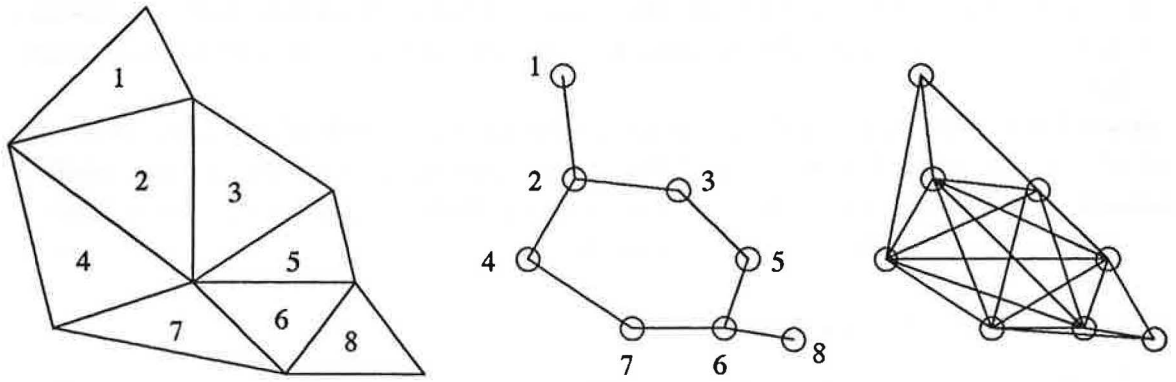

Figure 1: *A simple finite element mesh and its associated graphs. The middle figure is the edge communication graph and the one on the right is the true communication graph.*

The graph bisection algorithm uses just the connectivity information to bisect the mesh. This information can be represented as the connectivity (or communication) graph of the mesh. In Figure 1 we show a simple mesh and the corresponding communication graphs for the two cases of *edge* and *true* communication [5]. Vertices in the graph representation are associated with elements in the mesh. Links in the graph show the connectivity of the mesh. Thus, because element 1 is adjacent to element 2 there is a link between vertex 1 and vertex 2 in both graph representations. In the case of the edge communication graph, we say that two elements are connected if they share a common edge (or in three dimensions, a common face). Another possibility is to say that two elements are connected if they have any node in common. This gives the true communication graph, which has many more links than the edge graph, as can be seen in Figure 1. Which representation is most appropriate will depend on the details of the calculation that is to be performed on the mesh.

To bisect the graph, we first try and find the two extremal points which define the greatest diameter of the graph. In the case of the graph in Figure 1 these are the vertices 1 and 8. These points, or good approximations to them, can be found by selecting any starting point and labeling levels out from it

until all vertices are labeled. The last point to be labeled is then used as a new starting point and the process repeated. After a few such iterations the start and end vertices can be taken as the extremal points.

The basic step of the graph partitioning algorithm is then to divide the vertices of the graph according to their level numbers from the last seed point. In the case of the edge communication graph in Figure 1, if vertex (1) was the seed point then (2) would be level 2, (3,4) level 3, (4,6) level 4, (7) level 5 and (8) level 6. The bisection would then assign (1,2,3,4) to domain 1 and the rest to domain 2.

The `graph` option within the `partition` command selects this method. There is also an option, `cgraph`, to select whether the edge or true communication graph should be used.

4.4 Bandwidth minimisation methods

This method was proposed in [7] by Malone and is based on using a bandwidth minimisation algorithm which renumbers the vertices of a graph. If we define a connection in the graph as C_{ij} if it connects vertex i to j, then a bandwidth minimisation will relabel vertices so as to minimise the largest value of $|i - j|$ over all the graph.

The Malone method first requires a bandwidth minimisation on the nodal numbering. There are a number of well known algorithms to do this and we use the Gibbs, Poole, Stockmeyer or Gibbs, King variations [8]. Then the list of elements is sorted according to the lowest node number within each element. The partition into p parts is then made be dividing this ordered list into the required number of sections.

A simple variation on this method is to use an algorithm for *profile* minimisation, rather than bandwidth minimisation. It is shown in [9] that in many cases this gives slightly superior results to bandwidth minimisation. Another possibility is to use the Malone method in a recursive bisection manner, though this is not implemented in this release.

4.5 Kernighan and Lin methods

The Kernighan and Lin algorithm (KL) [6] can be used to improve an existing partition of a mesh. Like graph bisection, the KL method works in terms of the communication graph, rather than the actual mesh.

The optimal bisection of the mesh is viewed as finding the mapping of vertices to domains 1 and 2 which minimises the number of cut edges in the graph. A cut edge is a link in the graph with its end vertices in different domains. This criteria is not identical to that used in the cost geometric bisection and glutton algorithms, of minimising the number of interface nodes.

Given an existing partitioning of the vertices into 2 domains, the KL method first requires an evaluation of the net cost of swapping each vertex from its current domain to the other domain. This cost is the change in the number of cut edges. In Figure 2 we show a possible initial assignment for the vertices and the corresponding costs for each vertex if its domain assignment were to be swapped. A reduction in the number of cut edges is taken to be positive.

One step in the first KL iteration involves:

1. Find the domain with most vertices (take the first one if equal).

2. Find the vertex in this domain with the best gain (i.e. gives the greatest reduction in the number of cut edges if swapped). In the example this is vertex 2 (or 6).

3. Perform the swap of this vertex and update the cost values for neighbouring vertices, and record the current cost.

4. Mark this vertex as being locked, i.e. it will not be swapped again on this pass.

Then another, unlocked, vertex is transferred from the domain 2 to 1 in the same way. This process is repeated until all the vertices have been locked, even if at some stage the swaps actually increase the total number of cut edges in the graph. The reason for accepting such moves is that at the end of one pass the algorithm "back tracks" to find the point at which the edge cut cost was lowest. This point might occur after a number of "bad" moves have been made and allows the method to perform limited hill-climbing in its search for the global minimum.

After one such pass as described above, all the vertices are unlocked with domain assignments as at the lowest cost point. The process is repeated over again until there is a pass which provides no reduction in the number of cut edges.

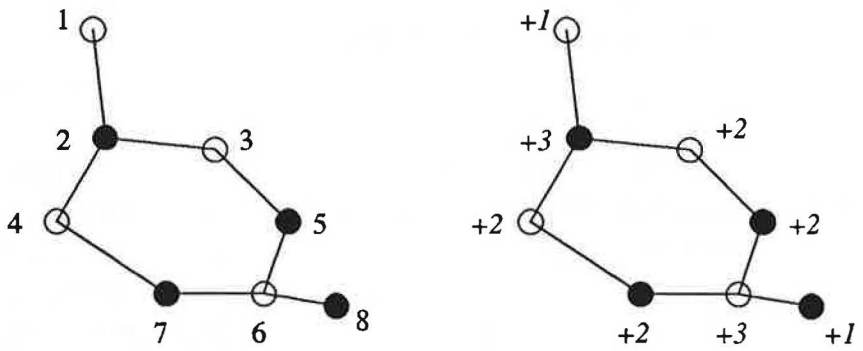

Figure 2: *The diagram on the left shows the graph of Figure 1 with half the vertices assigned to domain 1 (white circles) and half to domain 2 (black circles). The diagram on the right shows the reduction in cut edges for swapping each of the vertices.*

In implementing the KL algorithm it is important to perform the update of the cost list efficiently after each transfer to avoid the method becoming excessively time consuming. This has been achieved by using a linked list to store the cost values in, and by only adjusting neighbouring vertices.

The KL method requires a starting partition and there are a number of ways in which this can be generated. Currently three variations have been implemented, as follows:

- *Random start* (`kl-rand`). A completely random assignment of vertices to domains can be made. This method will give a different result each time it is run (depending on the random number generation), but does not tend to force the result in any particular direction. It is of course possible to perform several runs and select the best one with this method. This method is usually slower than the non-random approaches.

- *"Greedy" start* (`kl-greedy`). Since the basic KL method produces an ordered list and then always makes transfers from the domain with most nodes to the one with least, it is possible to start from a configuration in which all the elements are in one domain and just continue to make moves until balance is achieved. From this point the KL methods proceeds as normal. This is

similar to a "greedy" algorithm, just selecting those vertices that appear to be best in the first pass.

- *MinGraph* (`kl-rgb`). This is a similar approach to that used by Hu and Blake [2] whereby the graph bisection method is used to generate a starting configuration.

All these variations can be run as bisection methods, which is the default. However, it is not necessary to do so, as one can partition into unequal parts, e.g. splitting $1/3 : 2/3$ on the first cut and $1/3 : 1/3 : 1/3$ on the second. This option is available and allows non-power of 2 partitionings to be made, though results are generally not as good as for the bisection approach. To select the non-bisection version of any of the KL methods it is necessary to set the option `klbisc=false` in the `partition` command.

5 Modelling parallel system performance

For a given algorithm and data distribution it is often possible to specify how much data needs to be exchanged and between which processors this has to occur. The time to send n bytes between two processors in a parallel machine can often be approximated by the equation

$$t = t_{start} + nt_{send} \tag{2}$$

where t_{start} is the time to initialise communications and t_{send} is the time to send one byte. This formula ignores many factors, such as contention (due to other messages in the system), multihop costs, message length dependent buffering strategies and so on. However it is a reasonable first approximation and the parameters have been measured for most parallel systems in common use.

If the knowledge of the algorithm communication requirements is combined with the data distribution of a given partition and a communication cost model, like (2), then it is possible to estimate the actual communication time.

As a first step in this direction, 2 simple cost modelling schemes have been built into *ralpar*. There are two commands associated with these:

- `Machine` - this command allows machine communication parameters to be stored and viewed. The parameters used at present are the t_{start} and t_{send} used above, expressed in units of microseconds. R_∞ and $N_{1/2}$, which are the maximum communication rate and half performance message length respectively, are calculated from t_{start} and t_{send} stored, though these are not used. The `Machine` command also selects which machine type is to be used by the next `table` calculation.

- `Table` - this command calculates or displays the results for a given mesh using a range of partitioning methods. Results can also be written to file using this command.

The cost models currently implemented assume a simple matrix assembly calculation where each processor will have to transmit or receive data proportional to the number of interface nodes that it has. We assume that one 8 byte value will be associated with every interface node. For the first model we assume communications on a bus type architecture, such as a network of workstations connected via Ethernet. This is referred to as `SEQCOMM` as in this case only one message can be active at a time and

the total time spent in communication is just the sum of all the communication times for individual subdomains. Thus we calculate the communication time T_{comm} as,

$$\begin{aligned} T_{comm} &= \sum_{i=1}^{p} (8t_{send}I_i + N_i t_{start}) \\ &= N_T t_{start} + 8t_{send} \sum_{i=1}^{p} I_i \end{aligned} \tag{3}$$

where I_i is the number of interface nodes associated with subdomain i and N_i is the number of neighbour subdomains for i. N_T is the total number of neighbours, given by summing N_i. Clearly, there are a number of variations one could make in this formula, depending on the number of variables to be evaluated and which processors need to know the result. However, (3) is one possible form and we can make comparisons between methods using it, as long as we accept the results are qualitative rather than quantitative.

The second model that has been implemented is referred to as `PARACOMM`. This model allows the case of all processors to performing communication in parallel. It is assumed that each processor will have to send or receive 8 bytes of data for each interface node that belongs to its domain and also to start communication with as many other processors as it has neighbours. Hence the total communication time for one processor is just

$$T_{comm}(i) = 8t_{send}I_i + N_i t_{start} \tag{4}$$

Thus, communication tasks within a single processor will be done sequentially, but each processor will do this in parallel with all the others. On some systems, such as the Intel iPSC/860, it is possible to have concurrent communication on one processor using all the links available at once, though this may result in increased network contention. This possibility is ignored in the `PARACOMM` model. Note that in this model we use the I_i to represent *all* the interface nodes on subdomain i, where as in the sequential model interface nodes were assigned to the first subdomain they appeared in. Hence we have a larger amount of data interchange in this case.

Assuming that all the processors can operate in parallel without any contention, then the total communication time will be given by the largest value of $T_{comm}(i)$. The `PARACOMM` model reports both the maximum and average values of $T_{comm}(i)$. This model does not allow for any connection between the messages, which will be likely on many real parallel machines, so is again only qualitative.

The Examples section shows how the `Machine` and `Table` commands can be used. Future extensions will add to the number of models available.

6 Examples using *ralpar*

To start the standard version of *ralpar*, you need to be in a directory that contains the command definition file `commands.cmd`. If this file is not present, *ralpar* will stop and issue an error message. The RALBIC neutral file that describes the mesh to be partitioned should be copied (or linked) into the same directory.

The partitioner is started by issuing the executable name, if it has been placed in your search path, or the full pathname otherwise, e.g. `../ralpar`. This will cause a short introductory message to be printed followed by the prompt `Ralpar:`. Any valid command can than be issued and these commands are fully described in Appendix A. To get an online list of command names you just type

`help`. To get details of parameters for a specific command, such as the `input` command, you can use `help input`. The system is reasonably simple to use and working through one or two of the examples below should enable one to get to grips with it. The meshes used in these examples are distributed with *ralpar*.

6.1 Partitioning a two dimensional mesh

The file `CMPLX1.MSH` contains the RALBIC neutral file description of a two dimensional mesh with 165 quadrilateral elements and 200 nodes. Note that all mesh files have the extension `.MSH` and that the filename is in uppercase. The `input` command will automatically add the extension and convert the name to upper case. Thus to read in this file we can use the command

```
Ralpar: input cmplx1
```

which is the same as

```
Ralpar: input file=cmplx1 type=ralbic access=formatted
```

since the last two parameters are optional and default to these values.

If you have a version of *ralpar* with the GKS interface for two dimensional mesh display, you can than view the mesh using the command

```
Ralpar: display
```

This should give a plot of the mesh as shown in Figure 3.

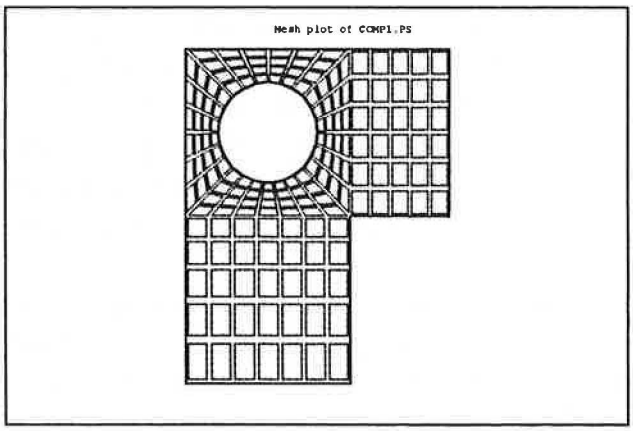

Figure 3: *The mesh* `CMPLX1.MSH` *as displayed using the* `display` *command.*

To partition the mesh the `partition` command is used. Thus, after having read the mesh with the `input` command, one can type

```
Ralpar: part 4 geo
Ralpar: display
Ralpar: part 4 band
Ralpar: display
```

to compare the results of two of the methods for the case of dividing the mesh into parts. These results are shown in Figures 4 and 5.

Note that after each `partition` command has been completed, the tool provides some information on the number of interface nodes that have been generated, such as:

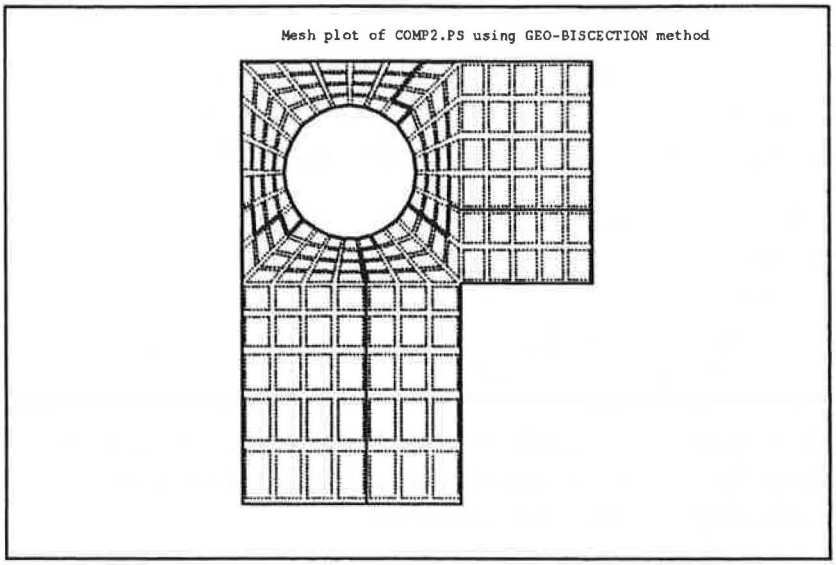

Figure 4: *The mesh* `CMPLX1.MSH` *partitioned into 4 using the geometric bisection method.*

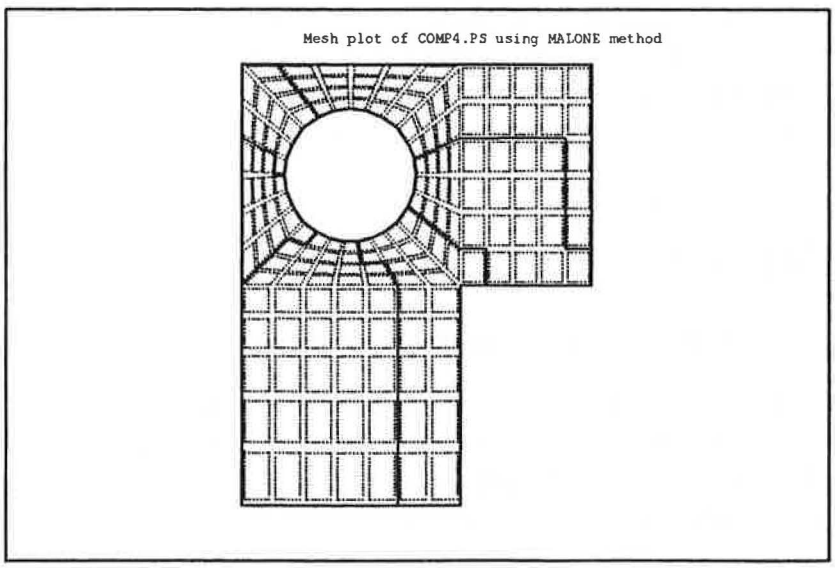

Figure 5: *The mesh* `CMPLX1.MSH` *partitioned into 4 using the bandwidth (Malone) method.*

```
 Inform: Interface node cost=     45
         Neighbour domains: Ave.=    2.500  Max.=    3  Min.=    2
 Inform: CPU time =     0.010 s
```

Also reported are the average number of domains about a domain and the minimum and maximum value of this quantity. For methods which work on the communication graph, such as the Kernighan and Lin ones, we also give the number of cut edges generated:

```
Ralpar: part 4 kl-gre
 Inform: Edge cut cost=      26 for     4 domains
 Inform: Interface node cost=     29
         Neighbour domains: Ave.=    2.500  Max.=    3  Min.=    2
 Inform: CPU time =     0.080 s
```

Note that the cut edge cost depends on the definition of the communication graph that is used. The default is to use the edge communication graph (described in the section on methods). Using the true communication graph with the same method, we get:

```
Ralpar: part 4 kl-gre cg=true
 Inform: Edge cut cost=      62 for     4 domains
 Inform: Interface node cost=     28
         Neighbour domains: Ave.=    2.500  Max.=    3  Min.=    2
 Inform: CPU time =     0.130 s
```

An important point to note here is that most optional parameters are *retained*, that is if you change the default in one command, it will become the new default. Thus any `partition` commands issued after the above one will use the true communication graph until it is explicitly changed.

Having made a partition of the mesh, the results can be written out to a new RALBIC neutral file using the `output` command. This creates a new version of the neutral file in which the partition number of each element is written in the element topology section. Thus to write the last partition generated to the file `CMPLX2.MSH` one would issue the command:

```
Ralpar: output cmplx2
```

Appendix C explains how RALBIC format files can be convert from or to other formats.

6.2 Partitioning a three dimensional mesh

All the same commands used for two dimensional meshes can be used the three dimensional ones, with the exception of the `display` command. The graphics are, at present, limited to display of two dimensional meshes. However, the AVS version of *ralpar*, described in Appendix B, can be used to visualise three dimensional partitioning results.

A simple three dimensional mesh, just consisting of one plane of hexahedral elements in the form of a "T", is given in the neutral file `T.MSH` which is provided with the standard *ralpar* release. This mesh can be read and partitioned with a number of methods using the commands:

```
Ralpar: inp t
 Inform: Data file read: Nodes=  3402 Elements=  1600
Ralpar: part 64 costgeo
```

```
 Inform: Interface node cost=  1022
         Neighbour domains: Ave.=   4.844  Max.=   6  Min.=   2
 Inform: CPU time =    0.710 s
Ralpar: part 64 kl-greedy
 Inform: Edge cut cost=    732 for   64 domains
 Inform: Interface node cost=  1438
         Neighbour domains: Ave.=   4.344  Max.=   6  Min.=   2
 Inform: CPU time =    1.360 s
Ralpar: part 64 kl-rgb
 Inform: Edge cut cost=    604 for   64 domains
 Inform: Interface node cost=  1150
         Neighbour domains: Ave.=   4.844  Max.=   6  Min.=   2
 Inform: CPU time =    2.100 s
Ralpar: part 64 glutton
 Inform: Interface node cost=  1022
         Neighbour domains: Ave.=   4.844  Max.=   6  Min.=   2
 Inform: CPU time =    1.860 s
Ralpar: part 64 profile
 Inform: Interface node cost=  2942
         Neighbour domains: Ave.=   3.188  Max.=   6  Min.=   1
 Inform: CPU time =    2.680 s
Ralpar:
```

For this highly regular case the cost-geometric and greedy (or glutton) algorithms give the best results in terms of interface nodes.

If we wish to assign non-uniform weights to each element we can do this with the `weight` command. In this case all the elements are of the same type, so using the weighting on number of nodes (`weight nodal`) will have no effect. Instead we can use a file to specify the weights. This can be a normal ASCII file with one number to a line which gives the weights in sequence. For the `T` mesh there are 1600 elements, and if, as an example, we were to take the weight of element i as $w_i = 1 + i/1600$ then a suitable file would need to be created with the values 1.000625, 1.00125,...,2.0. If we assume such a file (`wfile`) has been written, then it can be used in the following way:

```
Ralpar: weight file wfile
 Using element weights from file:wfile
Ralpar: info high
Ralpar: part 2 glutton
 Inform: Target weight per partition =   0.120025E+04
         Min. weight =  0.119931E+04   Max. weight =   0.120119E+04
         Ratio (max. weight)/(ave. weight) =   0.100078E+01
 Inform: Using ISTART=  5
 Inform: Interface node cost=   100
         Neighbour domains: Ave.=   1.000  Max.=   1  Min.=   1
 Inform: Workspace used     26210 out of   2800000 I*4 words
         Minimum value of RALPAR_MEMORY_SF =  15
```

```
 Inform: CPU time =     0.730 s
```

Note that we have used the `information` command to increase the amount of detail that is reported in this case. Much of the extra detail is not usually wanted, though in this case is allows us to see how well the load balance has worked in the case with weighting. If highly non-uniform weighting are used it is possible to get poor load balancing results. All weights must be strictly positive.

6.3 Comparing methods using the `table` command

The `machine` and `table` commands can be used to compare partitioning methods. Currently the `table` command only implements the simple model for bus type communication architectures and the parallel model, as described in the previous section. However, this command may be extended in future version of *ralpar*.

Having read in a mesh file, such as `T.MSH` it is necessary to first select the machine parameters to be used in the calculation. This is done with the `machine` command. To see what machines are available you can use the command:

```
Ralpar: machine action=display

 Machine                 Tstart     Tsend      n-half      R-inf

i860                    175.000     0.360    486.000      2.800
ipsc/2                  612.000     0.360   1750.000      2.800
SuperNode              1200.000     1.340    895.000      0.710
Transputer                8.730     1.130      7.844      0.898
```

This gives names and parameters of the available machines.

It is possible to extend this list within the current run of *ralpar*. For example to add the parameters for PVM 2.4 running over a Ethernet at RAL the commands would be:

```
Ralpar: mach add new 1500 1.5 name=pvm24
```

This will cause `pvm24` to be added to the list of available machines. Times are in microseconds and communication rate in Mbytes/s. New machine names and parameters are not automatically saved between runs of *ralpar*, though it is possible to use the `write` option of the `machine` command to save the data and reload it in another run with the `read` option.

To compare selected methods over a range of partition numbers, one can then use the `select` option of the `machine` command followed by the `table` command, e.g.

```
Ralpar: mach select pvm24
Ralpar: table compute methods=(kl-rgb,prof) part=(2,4,8)
....
Ralpar: table disp data=seqcomm

Table for: Sequ. Comm. model
-----------------------------------
                  2             4             8
kl-rgb      3.984000E+03  1.646400E+04  4.838400E+04
prof        4.104000E+03  1.111200E+04  2.628000E+04
```

The table gives the total time in microseconds for the communication based on the model discussed previously. In this case the high cost of the start up times means that the `profile` method, which minimises neighbour domains, gives better results than the `kl-rgb` method which gives a smaller interface.

References

[1] HA Schwarz: "Uber einige Abbildungsauf-gaben", *Ges. Math. Abh.* 11 65–83 (1869).

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

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

[4] RF Fowler, BW Henderson, and C Greenough, "Initial Experiences in Porting a Three-Dimensional Semiconductor Device Modelling Program to the Intel iPSC/860", Rutherford Appleton Laboratory Report, RAL-92-090 (1992).

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

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

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

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

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

[10] CRI Emson, C Greenough, NJ Diserens and KP Duffy, "RALBIC - A Simple Neutral File for Finite Element Data: File Definition", RAL Report RAL-87-102, 1987.

A The *ralpar* Command Summary

This Appendix provides a summary of all the commands available within *ralpar*. These commands are listed below in alphabetic order, with full details of each in the corresponding section.

A.1 CHANGE (Internal) - to change working directory

Syntax

CHAnge [DIRectory=<*string*>]

Description

The CHANGE command allows the user to change the current working directory without leaving the program.

Note that the CHANGE command is an INTERNAL command and is system dependent.

Parameters

DIRECTORY reset *string* : initial = '.'

DIRECTORY specifies the directory name to which the context should be changed. This must be a valid name for the host system.

Examples

Some examples on UNIX systems are:

```
Ralpar: change /u/cg/ralpar/tests
Ralpar: CHA ../tests
```

A.2 COPY (Internal) - to copy a file

Syntax

COPy [FILe1=<*string*>] [,FILe2=<*string*>]

Description

The COPY command allows the user to copy one file to another without leaving the program.

Note that the COPY command is an INTERNAL command and is system dependent.

Parameters

FILE1 required *string*

The FILE1 parameter specifies the source file. FILE1 must be a valid file name or expression on the host system.

FILE2 required *string*

The FILE2 parameter specifies the target file or directory. FILE2 must be a valid file name or expression on the host system.

Examples

Some examples on UNIX systems are:

```
Ralpar: COPY /u/cg/ralpar/tests/test1 data
Ralpar: cop ../tests/example .
```

A.3 DELETE (Internal) - to delete (remove) a file

Syntax

DELete [FILe=<*string*>]

Description

The DELETE command allows the user to delete files from the file system without leaving the program.

Note that the DELETE command is an INTERNAL command and is system dependent.

Parameters

FILE required *string*

FILE specifies the file(s) to be deleted. FILE must be a valid file name or expression on the host system.

Examples

Some examples on UNIX systems are:

```
Ralpar: DELETE data
Ralpar: del ../tests/example
```

A.4 DISPLAY (Application) - to display partition results

Syntax

Display [Boundary=<*choice*>] [,Mesh=<*choice*>] [,Label=<*choice*>]

Description

The DISPLAY command display the current mesh and partition on the selected output device. The finite elements in the mesh a shrunk to help see the partition boundaries. The partition boundaries are shown in dark lines and the elements are displayed with broken lines. (This output is only currently available for two-dimensional meshes.)

Parameters

BOUNDARY retained *choice* : initial = YES

The BOUNDARY parameter controls the display of the partition boundaries. Its values are: NO (no boundaries) and YES (partition boundaries are displayed).

MESH retained *choice* : initial = YES

The MESH parameter controls the display of the mesh. Its values are: NO (no mesh) and YES (mesh displayed).

LABEL retained *choice* : initial = NO

The LABEL parameter controls the display of the node and element numbers.Its values are: NO (no labels) and YES (node and element numbers displayed).

Examples

```
Ralpar: DISPLAY
Ralpar: dis yes yes yes
```

A.5 HELP (Internal) - to access HELP system

Syntax

Help [KEY=<*string*>] [,OPTion=<*choice*>]

Description

Accesses to the inbuilt HELP system within the command decoder. HELP is one of the internal commands of the command processor and has a companion command SYNTAX.

HELP has two parameters allowing the selection of help on a specific command and the level of help required (SUMMARY, BRIEF and SYNTAX). If no command name is given summary help is given on all the commands currently defined.

If an ambiguous or invalid command name is given a warning or error message is given.

BRIEF help gives information on the purpose, syntax and the current state of the selected command. A table of command keywords, their type, status and current value (if applicable) is printed.

Parameters

KEY reset *string* : initial =

Either the global command name SUMMARY, or the specific command name on which help is sought.

OPTION reset *choice* : initial = BRIEF

The level of help required. This can be SUMMARY, BRIEF or SYNTAX.

Examples

```
 Ralpar: help plist

Name     : PLIST

Purpose : to list element numbers in a partition
Syntax   : PList [Partition=<integer>] [,Limit=<integer>]


Keyword        Type             Status        Current Value
-------------------------------------------------------------

PARTITION      integer          retained       0
LIMIT          integer          retained       50
```

A.6 INFORMATION (Application) - control message output

Syntax

INFormation [LEvel=<*choice*>]

Description

The INFORMATION command allows the user to control the amount of information provided by the program during its execution.

Parameters

LEVEL retained *choice* : initial = QUIET

The LEVEL parameter controls the level of information output. Its values are: QUIET, MEDIUM, HIGH and VERBOSE.

Examples

```
Ralpar: INFORMATION VERBOSE
Ralpar: info q
```

A.7 INPUT (Application) - to read in mesh file

Syntax

Input [File=<*string*>] [,Type=<*choice*>] [,Access=<*choice*>]

Description

The INPUT commands reads a complete mesh from the specified file. The description contains the nodal positions and the element topologies. The command allows for the three input formats currently being used: FELIB, BERTIN and RALBIC.
Provision has been made for both formatted and binary data files.

Parameters

FILE required *string*

The FILE parameter specifies the file name in which the data is held.

TYPE retained *choice* : initial = RALBIC

The TYPE parameter specifies the type of format in which the data is stored. It has values: FELIB, BERTIN and RALBIC.

ACCESS retained *choice* : initial = FORMATTED

The ACCESS parameter specifies whether the data is stored in a formatted or binary format. It has values: FORMATTED and BINARY.

Examples

```
Ralpar: INPUT FILE=DATA, TYPE=FELIB, ACCESS=FORMATTED
Ralpar: i data
```

A.8 LIST (Internal) - to provide directory listing

Syntax

LISt [FILe=<*string*>]

Description

The LIST allows the user to list files available to him on the systems file store.

Note that the LIST command is an INTERNAL command and is system dependent.

Parameters

FILE reset *string* : initial =

The FILE parameter specifies a file name or file mask over which the listing is to search. (*The file mask will be system dependent*).

Examples

Some examples on UNIX systems are:

```
Ralpar: LIST FILE=data
Ralpar: lis "*.f*"
```

A.9 MACHINE (Application) - to specify machine constants

Syntax

MAChine [ACtion=<*choice*>] [,Type=<*choice*>] [,TSTART=<*real*>]
[,TSEND=<*real*>] [,NAme=<*string*>] [,FILename=<*string*>]

Description

The MACHINE command allows the user control and modify the machine constant table.

The command allows the use to read his own set of machine constants into the table and to save any modification made to these values.

Modification can include the addition of new news and changing the parameter values associated with a particular machine. Once modified the new constants table can be written to disc.

Parameters ACTION retained *choice* : initial = SELECT

The ACTION parameter control the process of the command. The current actions are:

SELECT - To select a particular set of machine values
READ - To read a table of values from FILENAME
WRITE - To write the machine constant table to FILENAME
DISPLAY - To display the current table values
ADD - To add a new machine to the table

TYPE retained *choice* : initial = I860

The TYPE parameter specifies which set of machine parameters are to be used. The current default systems are i860, iPSC/2, SUPERNODE and TRANSPUTER.

TSTART retained *real* : initial = 175.0

The TSTART parameter specifies the communication start-up time in seconds.

TSEND retained *real* : initial = 0.36

The TSEND parameter specifies the communication rate in bytes per second.

NAME reset *string* : initial =

The NAME parameter specifies the name of the new machine to be added to the machine constants table.

FILENAME retained *string* : initial = machine.cst

The FILENAME parameter specifies the filename where the machine constants table is stored.

Examples

```
Ralpar: MACHINE TYPE=SUPERNODE ACTION=SELECT
Ralpar: mach add new 1500 1.5 pvm
Ralpar: machine action=dis
```

Machine	Tstart	Tsend	n-half	R-inf
i860	175.000	0.360	486.000	2.800
ipsc/2	612.000	0.360	1750.000	2.800
SuperNode	1200.000	1.340	895.000	0.710
Transputer	8.730	1.130	7.844	0.898
paragon	175.000	0.360	486.000	2.800

A.10 OUTPUT (Application) - to write neutral file with partition numbers

Syntax

Output [FIle=<*string*>] [,Type=<*choice*>] [,Access=<*choice*>]

Description

The OUTPUT command writes a complete mesh from the specified file. The description contains the nodal positions, element topologies and domain identification. The command allows for the three input formats currently being used: FELIB, BERTIN and RALBIC. Provision has been made for both formatted and binary data files.

Parameters

FILE retained *string* : initial = IDENTIFY

The FILE parameter specifies the file name in which the data is held.

TYPE retained *choice* : initial = RALBIC

The TYPE parameter specifies the type of format in which the data is stored. It has values: FELIB, BERTIN and RALBIC.

ACCESS retained *choice* : initial = FORMATTED

The ACCESS parameter specifies whether the data is stored in a formatted or binary format. It has values: FORMATTED and BINARY.

Examples

```
Ralpar: OUTPUT FILE=IDENTIFY TYPE=RALBIC ACCESS=FORMATTED
Ralpar: out data900 bertin binary
```

A.11 PARTITION (Application) - to partition the data

Syntax

Partition [Processors=<*integer*>] [,Method=<*choice*>]
[,LEvel=<*integer*>] [,CGraph=<*choice*>]
[,KLBISC=<*choice*>] [,PWeight=<*real_list*>]
[,FIlepw=<*string*>]

Description

The PARTITION command controls the partitioning of the meshes read into the program. The command specifies the number of partitions, the method to be used and the values of some control parameters.

Parameters

PROCESSORS retained *integer* : initial = 4

The PROCESSORS parameter specifies the number of partitions into which the mesh is to be partitioned. The number of partitions must be compatible with the method being used.

METHOD retained *choice* : initial = GEO-BIS

The METHOD parameter specifies the partitioning method to be used. The current methods are:

GEO-BIS - geometric bisection
COSTGEO - geometric bisection using lowest cost directions
GREEDY - greedy algorithm due to Farhat
GLUTTON - greedy algorithm with multiple seed point searching for minimum cost
BANDWTH - nodal re-ordering for minimum bandwidth (Malone)
PROFILE - nodal re-ordering for minimum profile width
INERTIA - bisection based on axes of inertia
R-INER - recursive inertial bisection
KL-GREEDY - Kernighan & Lin with GREEDY starting point
KL-RAND - Kernighan & Lin with random starting point
SPEC - recursive spectral bisection
GRAPH - recursive graph bisection
KL-RGB - Kernighan & Lin with RGB starting point

LEVEL retained *integer* : initial = 5

The LEVEL parameter specifies the number of levels to be searched in the GLUTTON algorithm.

CGRAPH retained *choice* : initial = EDGE

The CGRAPH parameter specifies the type of communication to be used in generating the communication graph. Possible values are EDGE or TRUE.

KLBISC retained *choice* : initial = TRUE

The KLBISC parameter specifies whether the Kernighan & Lin methods are to be recursive bisection methods. Possible values are TRUE or FALSE.

PWEIGHT reset *real_list* : initial = 0

The weights of each partition may be set by the PWEIGHT parameter. All weights must be positive. If neither PWEIGHT or PFILE are specified, all partitions will have the same weight.

FILEPW reset *string* : initial =

Partition weights will be read from the file named FILEPW if this parameter is set. The file is read in free format and should contain one number per line giving the weight of each partition in sequence. All values must be greater than 0.

Examples

```
Ralpar: PARTITION PROC=16 METHOD=KL-RAND CGRAPH=TRUE
Ralpar: par 32 geo
Ralpar: part 6 prof pweight=(4,2,2,1,1,1)
```

A.12 PLIST (Application) - to list element numbers in a partition

Syntax

PList [Partition=<*integer*>] [,Limit=<*integer*>]

Description

The PLIST command allows the user to display the elements allocated to a particular partition. The command allows the user to control which partition results are display and the number of elements displayed.

Parameters

PARTITION retained *integer* : initial = 0

The PARTITION parameter specifies which partition is to be displayed. If this is set to zero (0) all partition data will be displayed.

LIMIT retained *integer* : initial = 50

The LIMIT parameter allows the user to limit the amount of data displayed by the PLIST command.

Examples

```
Ralpar: PLIST PARTITION=2 LIMIT=10
Ralpar: pl 0 50
```

A.13 QUIT (Application) - to quit program

Syntax

Quit

Description

The QUIT command terminates the program and closes all external files.

Parameters

This command has no parameters.

Examples

Ralpar: quit

A.14 READ (Internal) - to redirect the input stream to read from a file

Syntax

REad [FIle=<*string*>] [,ECHO=<*choice*>]

Description

The READ allows the user to process a stream of program commands stored in a file on the system. The commands is the file are process as standard commands and requests for information and error reports are directed to the terminal.

Note that the READ command is an INTERNAL command and is system dependent.

Parameters

FILE required *string*

Input file name containing program commands.

ECHO reset *choice* : initial = OFF

Echo control option. Values can be ON or OFF.

Examples

```
Ralpar: READ FILE=script, ECHO=ON
Ralpar: read long-run off
```

A.15 RENAME (Internal) - to rename a file

Syntax

RENname [FILe1=<*string*>] [,FILe2=<*string*>]

Description

The RENAME command allows the user to rename files from the file system without leaving the program.

Note that the RENAME command is an INTERNAL command and is system dependent.

Parameters

FILE1 required *string*

The FILE1 parameter specifies the file to be renamed. FILE1 must be a valid file name or expression on the host system.

FILE2 required *string*

The FILE parameter specifies the target file. FILE2 must be a valid file name or expression on the host system.

Examples

```
Ralpar: rename xyz.msh XYZ.MSH
```

A.16 SYNTAX (Internal) - to provide the syntax of a command

Syntax

SYNtax [COMmand=<*string*>]

Description

Displays the formal syntax of all the currently defined commands. If the syntax of a specific command name is required then that name is given as a parameter to the command.

Parameters

COMMAND retained *string* : initial = ALL

Specifies the commands name for which the syntax is required. If the syntax of all the currently defined commands is required, then the special command name ALL should be used.

Examples

```
Ralpar: syntax machine

 MAChine [ACtion=<choice>] [,Type=<choice>] [,TSTART=<real>]
         [,TSEND=<real>] [,FILename=<string>] [,NAme=<string>]
```

A.17 TABLE (Application) - to evaluate or display table of costs

Syntax

TABle [ACtion=<*choice*>] [,MEthods=<*string*>]
[,PARtitions=<*string_list* >] [,FILename=<*integer_list* >]
[,DAta=<*choice*>]

Description

The TABLE command allows the user to compare a range of methods over a number of partitions. The user has to define the methods to be used, and the number of partitions which are used in the COMPUTE action. Results can then be viewed with the DISPLAY action or written to file with WRITE.

Parameters

ACTION retained *choice* : initial = COMPUTE

The ACTION parameter specifies the action to be taken by the TABLE command. Currently values are:

COMPUTE - to perform the cost table calculations

DISPLAY - to display the calculated cost table

WRITE - to write a calculated cost table to file

FILE retained *string* : initial = TABLE.CNST

The FILE parameter specifies the file name of the cost table.

METHODS retained *string_list* : initial = (GEO-BIS,INERTIA)

The METHODS parameter specifies the methods to be used in the table calculations. Available methods are: GEO-BIS, COSTGEO, GREEDY, GLUTTON, BANDWDT, PROFILE, INERTIA, R-INER, KL-GREEDY, KL-RAND, SPEC, GRAPH and KL-RGB.

PARTITIONS retained *integer_list* : initial = (2,4)

The PARTITIONS parameter specifies the partitions that are to be used in calculating the cost table.

DATA retained *choice* : initial = SEQCOMM

The DATA parameter specifies which results are to be shown when ACTION=DISPLAY is used. Options available are: ALL, SEQCOMM, PARACOMM, INTERFACE, NEIGHBOURS. Note that all the data is calculated when ACTION=COMPUTE, irrespective of the setting of this option.

Examples

```
Ralpar: TABLE ACTION=READ FILE=COST-TABLE
Ralpar: table methods=(geo-bis,costgeo,greedy) part=(2,4,8,16)
```

A.18 WEIGHT (Application) - to define element weights

Syntax

WEight [MEthod=<*choice*>] [,File=<*string*>]

Description

The WEIGHT command allows the user to give weighting to the elements of a mesh being partitioned. The weight of elements can be used to achieve load balance in applications where mixed element types are being used.

The WEIGHT command will also allow the use to read weighting information from a file.

Parameters

METHOD retained *choice* : initial = UNIFORM

The METHOD parameter specifies the weighting method to be used. Possible values are:

UNIFORM - all elements have the same weight

NODAL - elements are weighted according to the number of nodes they contain

FILE - the weighting information is to be input from a file

FILE retained *string* : initial =

The FILE parameter specifies the file from which weighting information is to be taken.

Examples

```
Ralpar: WEIGHT METHOD=UNIFORM
Ralpar: we file weighting-info
```

A.19 WRITE (Internal) - to provide monitoring of a session

Syntax

WRIte [STAte=<*choice*>] [,FIle=<*string*>] [,PROmpt=<*choice*>]

Description

Redirects the command decoder echo output to the file specified by the FILE parameter. The information flow is controlled by the STATE parameter. This command can enable the constructions of command files to drive the program in a background mode.

The echoing of the command prompt can be controlled using the PROMPT parameter.

Parameters

STATE required *choice*

Controls the flow of information to the monitoring file It has values ON, OFF or CLOSE. ON switches on monitoring. OFF suspends it but does not close the file and CLOSE ends monitoring and closes the file.

FILE retained *string* : initial = MONITOR

Output file name to receive the monitoring stream.

PROMPT reset *choice* : initial = OFF

Allows you to select whether the command prompt is echoed in the monitoring file. It has values ON or OFF.

Examples

In this example commands are written to the file MONITOR without the prompt:

```
Ralpar: WRITE STATE=ON FILE=MONITOR PROMPT=OFF
```

B The *avsralpar* interface

This Appendix gives a brief overview of the AVS version of the partitioner, *avsralpar*. It assumes that the user has at least a basic knowledge of AVS and how to run it on their system.

Avsralpar allows graphical display of the effects of partitioning. It does not at present allow you to write out the results of the partitioning - for this you have to run the command line version of *ralpar*. Though the AVS version does allow the maximum mesh size to be altered in the same way as the command line version, *avsralpar* may not be able to deal with very large meshes due to the extra memory needs of AVS.

B.1 Starting up *avsralpar*

You can start the AVS version by reading the network file `<avsralpar>/Networks/ralpar` into the AVS networks editor, where `<avsralpar>` is the directory containing *avsralpar* on your system. Alternatively this network can be run directly using:

```
avs -network <avsralpar>/Networks/ralpar
```

For convenience this command is often aliased to `avsralpar`.

Similar information to that provided by the command line version is written on the standard output. This includes interface node cost, neighbour information and cut edge counts.

When run without the Network editor, *avsralpar* will create two windows. The one on the right is the geometry viewer which displays the mesh. The control panel appears on the left hand side. By default the tool is set up to read the mesh file `T.MSH` on start up.

The partitioned mesh can be examined in the geometry viewer using the standard AVS control techniques.

B.2 The control panel commands

There are five options available on the "Top level stack" in the control panel. These are as follows:

- *ucd to geom.* This gives two dials to input the element shrink factor (0-100) and the explode factor (0-100). The first controls the reduction in size of element faces, to enable the mesh detail to be seen. The second controls the separation of materials, as different domains are displayed as different materials.

  By default only the external element faces of each domain are shown. `External edges` causes only the external edges to be shown. `All faces` requests AVS to show all the mesh. This is not recommended for large meshes.

- *ucd cell color.* This option has no controls at present.

- *generate colormap.* This can control the colouring of the different domains. Domains are coloured according to their domain number. The default is a linear colour map with red corresponding to domain 1 and blue to the last domain. Normally there is no need to alter this.

- *animated integer.* This module can be used to step through a range of numbers of domains using a given method. This is mainly for display purposes and works best with small meshes using one of the non-bisection methods. For example, if `min value` is set to 1, `max value` to 64,

and `steps` to 64, then clicking on `sleep` to turn it off will cause the network to step through all the results from 1 to 64 domains.

- *partitioner*. This selects the partitioner module. The `Identify Browser` selects the mesh file to read. These must be in RALBIC format and have names of the form `FILE.MSH`. Click on the file you wish to read. On the bottom left is the list of methods. Clicking on one of these causes the method to be used on the current mesh. The number of partitions to be generated is given by the `partitions` dial. The `EGraph` and `KLBisc` buttons can be toggled on and off. They do not cause a new partition to be generated, but only effect the result of subsequent calculations. `EGraph` on implies that the methods that use the connection graph will assume that only elements with a common face are connected, otherwise any elements with one or more common nodes are connected. `KLBisc` on implies that the KL methods work as bisection methods, otherwise they work on one domain at a time.

Other useful options are available under the `Geometry Viewer` menu of the `Data viewers` button. On machines with limited graphics hardware it may be useful to enable the bounding box, so that AVS does not attempt continuous image updates during rotation.

Within the Geometry viewer, all the subdomains are separate objects so that they can be selected and examined individually if required.

C Interfacing to the RALBIC neutral file format

The RALBIC neutral file format [10] is a simple ASCII file format for storing details of finite element meshes and results. This file format permits many different data fields to be stored in a single file, but for *ralpar* only the element topology and nodal coordinates of the mesh are required.

By convention all RALBIC file names are in upper case and the extension `.MSH` is used. A simple example of the RALBIC neutral file for a two dimensional mesh with just two elements is shown below:

```
$HEAD
 VER_03_87
$CASE
  Partitio
$NODE
     6    2
     1  0.0000000D+00  0.0000000D+00
     2  0.0000000D+00  0.1000000D+01
     3  0.1000000D+01  0.0000000D+00
     4  0.1000000D+01  0.1000000D+01
     5  0.0000000D+00  0.2000000D+01
     6  0.1000000D+00  0.2000000D+01
$END-NODE
$ELEM
     2    8
     1 QU04            AIR    4    1    2    4    3
     2 QU04            AIR    4    3    4    6    5
$END-ELEM
$END-CASE
$END-HEAD
```

The node section of the file is quite simple and just contains x and y coordinates for each node. The element topology section (`$ELEM`) contains one record for each element. This gives the element number, the element type, the material type. number of nodes in the element and then the node number. The material type field is ignored on input into *ralpar*. On output, an integer is written into this field indicating the subdomain to which this element has been assigned.

To actually create such files from another format, two example programs are provided with the standard release of *ralpar*. The first is `loc2ral.f` which reads a mesh file in a simple format and writes out a corresponding RALBIC neutral file. The source code for this program is provided along with the RALBIC library so that it may be easily modified to read the data according to the preferred format of the user. This will then provide a tool to convert from any desired format to RALBIC format.

A similar program to convert the output RALBIC file from *ralpar* into a local format is also supplied as `ral2loc.f`. This can be modified to write the specific information that will be required for the separate subdomains. The element types that are supported by the RALBIC interface include those listed in Table 1.

For *ralpar* the ordering of nodes within an element is not important, even though the RALBIC format does specify the required order. For the AVS version, *avsralpar*, the display of elements may be incorrect if strange nodal orderings within elements are used, though the results should still be correct. The assumed scheme is to number the nodes anticlockwise. For 3D elements, the lower plane is numbered first, then the upper plane, starting at the same point as the first plane.

Element	No. of nodes	Name	Type No.
Hexahedra	8	BR08	1
Prism	6	WG06	2
Tetrahedra	4	TE04	3
Triangle	3	TR03	7
Quadrilateral	4	QU04	9
Line	2	LI02	11
Pyramid	5	PY05	5

Table 1: Some RALBIC element types.

D Control of memory allocation

Ralpar is written in standard Fortran 77 as far as possible. As the standard does not allow dynamic memory allocation, most arrays are treated as having fixed size. To allow you to adjust the memory use according to that available on your machine (and size of mesh), dynamic allocation of work arrays is made at the top level. By default, space is allocated for about 30000 nodes/elements. To partition larger meshes you need to set the environment variables `RALPAR_NODES` and/or `RALPAR_ELEMENTS`. For example, if you have 100000 nodes and 120000 elements you could issue the command (csh):

```
setenv RALPAR_ELEMENTS 130000
```

before running *ralpar*. In the Bourne shell the command would be:

```
RALPAR_ELEMENTS=130000
export RALPAR_ELEMENTS
```

Note that if you only set one of nodes or elements, the other defaults to the same value.

Of course, some methods require much more memory than other methods and the partitioner allocates enough for all methods by default (assuming "reasonable" limits on connectivity). You can change the amount of workspace allocated by setting the environment variable `RALPAR_MEMORY_SF` if your machine cannot allocate sufficient memory for a given number of nodes/elements. The default value of this is 100, so to half the workspace, you could issue the command (in csh):

```
setenv RALPAR_MEMORY_SF 50
```

Note that some methods may not run with this reduced workspace, and you should only try this if the partitioner will not run because it failed to allocate enough memory for the requested mesh size.

To find out how much of the available workspace is actually being used by each method, you can use the command `information high`. Any subsequent `partition` commands will then also print out some details of how much space was actually used.