Scalable Unstructured Mesh Decomposition

*K McManus, S P Johnson, C Walshaw, M Cross, P Chow**
k.mcmanus@gre.ac.uk

Parallel Processing Research Group,
The University of Greenwich, London, UK
* Fujitsu European Centre for Information Technology

Abstract

As the efficiency of parallel software increases it is becoming common to measure near linear speedup for many applications. For a problem size N on P processors then with software running at $O(N/P)$ the performance restrictions due to file i/o systems and mesh decomposition running at $O(N)$ become increasingly apparent especially for large P. For distributed memory parallel systems an additional limit to scalability results from the finite memory size available for i/o scatter/gather operations. Simple strategies developed to address the scalability of scatter/gather operations for unstructured mesh based applications have been extended to provide scalable mesh decomposition through the development of a parallel graph partitioning code, JOSTLE [8]. The focus of this work is directed towards the development of generic strategies that can be incorporated into the Computer Aided Parallelisation Tools (CAPTools) project [1].

1 Introduction

Mesh partitioning and distribution is ordinarily handled as either a pre-processing of the mesh file(s) or as a sequential i/o process. Both schemes are flawed. Pre-processing implies knowledge of the intended number of processors and a sequential machine with sufficient power and memory to accommodate the task. Sequential i/o is a performance bottleneck and generally requires that at least one processor has sufficient memory to hold the data required to describe the entire mesh. Unstructured mesh parallelisation has typically relied on buffered scatter/gather operations to implement the data renumbering required for file i/o functions. Such operations encounter memory scalability problems on multiple address space systems. Many of these difficulties may arguably be surmounted with the adoption of a single address space memory system (implemented either as hardware or software). It would be unfortunate however, to mandate a single address space and hence limit portability and/or performance simply to make it possible to handle mesh partitioning.

Clearly there are alternatives such as parallel mesh generation but the current state of the art has not reached a level that is acceptable to users. The user requirement is for an application code to read from file, an often painstakingly crafted mesh, and produce data files that may be post-processed and further analysed. A scalable solution is therefore required to replace conventional scatter/gather operations and to localise mesh partitioning.

Key to the implementation of scalable mesh partitioning has been the development of the parallel graph partitioning code, JOSTLE [8], which provides a rapid, highly localised partition optimisation algorithm. This has enabled localisation and hence scalability of the entire mesh decomposition process. A primary motivation of this development has been the Computer Aided Parallelisation Tools (CAPTools) project [1] which provides an interactive environment for the semi-automatic transformation of application source code into parallel message passing source code. This requires the development of techniques that are sufficiently generic to be applicable to the potentially enormous range

of data structures used in unstructured mesh based applications.

2 Overlapping Domain Decomposition

Computational methods for continuum mechanics modelling require a geometric representation of the problem space. Discretisation of the differential equations used to model the continuum phenomena results in programs for which data dependencies across the discretisation are highly localised. This has naturally led to the adoption of geometric domain decomposition techniques for mapping Finite Volume and Finite Element based applications onto Distributed Memory (DM) parallel machines [6]. Inter-sub-domain data dependencies are resolved through communication of the necessary data from the sub-domain on which it is assigned, to the sub-domain on which it is accessed. For many applications it has proved to be convenient to extend each sub-domain to overlap with its neighbours allowing communicated data to be copied into the overlap layer and consequently requiring no alteration to the source code in order to address overlap data [4]. Localised data dependence means that the depth of the overlap layer, and hence the amount of data to be communicated, is kept to an acceptably low level.

2.1 Mesh partitioning

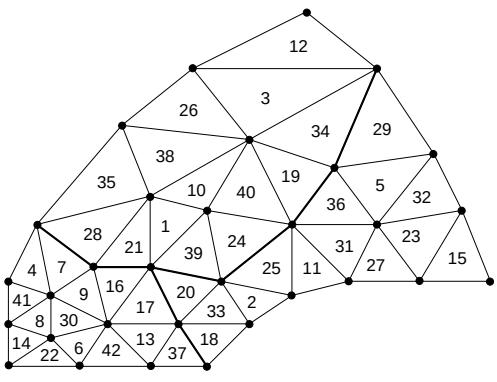

Figure 1: Simple unstructured mesh.

Consider the unstructured mesh in Figure 1 that has two mesh entity types, vertices and elements. The adopted mesh partitioning scheme is to select the mesh entity type that is associated with the largest amount of calculation, elements for example. A primary partition is calculated to exclusively allocate each entity of that type to a processor so as to allow a balanced distribution of the entities across all processors.

Loops that address data structures, and hence express data dependence, are extracted from the application code to provide inspector loops [2]. These inspector loops are used, at run-time, to construct (amongst other things) an undirected graph, $G(N, E)$ of nodes N and edges E, that accurately represents the data dependencies that are required by the code, and are defined by the mesh topology for the selected entity type. This graph (which may be weighted according to some workload prediction) is passed to a graph partitioning tool, such as JOSTLE (discussed further in Section 3.4). The graph partitioner attempts to partition the graph so as to minimise the imbalance between the processors and to minimise the number of graph edges cut between different sub-domains, where these cut edges will infer communications in the parallel code. The graph partition is returned as an 'ownership array' containing the sub-domain (processor) number that 'owns' each entity of the selected type. The secondary partitions that describe ownership of other entity types (e.g. vertices) are determined in accordance with the primary partition and the mesh connectivity.

The set of entities owned by a processor is referred to as that processor's core set. Parallel execution is achieved by performing calculations that relate to the assigning of values to a data structure representing a particular entity only on the processor that owns that entity (owner computes rule). This calculation may require access to non-owned data. For example, in a linear differencing code the evaluation of each element based variable requires the values from each connected (adjacent) element. This data dependence requires an element overlap as illustrated in Figure 2. Typically, a set of such non-owned data items will be required, and can be communicated in a single set of bulk communications. Each set of overlap entities therefore defines the components of a communication set.

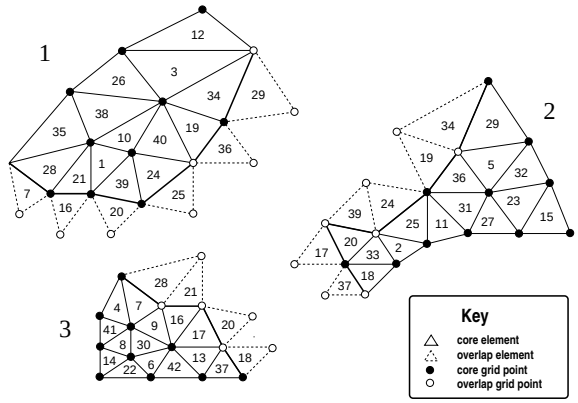

Figure 2: Partition of elements in a simple mesh onto three processors with overlaps.

2.2 Data localisation

In order to scale an application in memory each processor must store only the necessary data, that is it's subdomain core and overlap. For scalability to very large P all data must be local, global data structures are simply not possible. For an unstructured mesh code it transpires that this is largely straightforward and has the significant advantage of simplifying much of the parallelisation task. It has been shown that, for many codes, this strategy enables parallelisation of many subroutines with little or no modification to the source code [4]. Packing the data associated with each local mesh entity into local memory implies a local mesh numbering. Pointer arrays, used extensively to describe an unstructured mesh, will also be packed into local memory. For example, a structure that holds the vertex numbers for each element, will locally contain global vertex numbers for each locally numbered element. Global to local pointers are consequently required in inspector loops [5] used to renumber the contents of such local pointer arrays into local mesh numbering. But a global to local pointer would require a globally dimensioned array and so a scalable solution is required.

With locally renumbered sub-domains (core and overlap) on each processor, concurrent execution requires only the inclusion of communication calls into the code. With few exceptions [4], only two classes of communication are required in the body of the code; overlap update and global commutative. Overlap update copies data from each subdomain core, where it is assigned, into the appropriate neighbouring subdomain's overlap, where it is accessed. This operation scales well as it only requires communication between neighbouring subdomains (processors) and so, with a good partition, the communication cost tends towards being approximately constant for increasing numbers of processors P [3]. Global commutative (PVM reduce, MPI collective) performs some commutative arithmetic or logical operation (add, and, etc.) across data held on each processor. This is a potentially expensive communication as it requires communication amongst all processors but the use of binary tree (hypercube) communications restricts the latency cost increases to an acceptable $\log_2(P-1)$. Broadcast operations are a special case of global commutative.

3 Scalability

The focus of attention on scalability has tended to be directed towards computational scalability, the ability to profitably extend an application to greater numbers of processors. This provides a somewhat arbitrary limit resulting from a gradual decrease in parallel efficiency as P increases. Memory scalability is the degree to which a problem size can be increased with increasing P. The limit for memory scalability is a sudden cut–off point at which it is no longer possible to fit a problem into memory. This either causes the code to fail or initiates memory paging with a significant deterioration in efficiency. Both effects are highly system and application dependent.

3.1 Scalable Implementation

It has become accepted practice to present impressive, often superlinear, speed-up curves for parallel applications. But for practical applications the scalability issues associated with the input of data files, incremental output of time step variables and, perhaps most importantly for unstructured mesh applications, the decomposition and distribution of meshes, have not been fully addressed and present a significant parallel

overhead. These tasks are not readily amenable to parallelisation, file access hardware remains largely sequential and it is arguable that in a 'real' situation the mesh distribution overhead of a few minutes is eclipsed by run-times of hours or days. If, however, parallel hardware is to become interactive or user friendly the i/o scalability problems must be ameliorated.

3.2 Gathering data to file

Gathering of the resulting data sets to file necessitates re-ordering the data to global numbering. Scalable gathering is achieved by copying small quantities of data to the i/o processor in such a way as to ensure that the next data item to be copied to file is always available and easily located. Small blocks of contiguous local data, beginning with the first local entity number, are copied from every processor to the i/o processor, together with the global entity numbers (held locally) for the data block. As the local numbering is in global order, each block will be accessed sequentially and so a pointer records the current position in each block. A list of block numbers is sorted in accordance with the current entity number in each block. The next required data item is therefore always in the block which is first in the list, from where it, and all consecutively numbered items can be copied to file, incrementing the current position accordingly. The block number is now inserted back into the list at the appropriate position and so the first entry in the list will once again be the block containing the next required data item. To further improve efficiency, each block is maintained in two halves so that as a block becomes half empty the appropriate processor is sent a request for the next (half) block. This allows the communication of the data to the i/o processor to be concurrent with the file output operation and so ensure that the blocks never empty until all data has been written to file.

The method adapts easily to take advantage of whatever memory is available on the i/o processor(s) at run-time simply by increasing the block size to fit the available memory. Speed of operation is dependent on the file bandwidth, inter-processor communication performance, block size, mesh size N, available memory and number of processors P. Difficulties may arise with extremely large P, in which case the problem size will be much larger than the memory per processor resulting in many small blocks, heavy processor to i/o processor data traffic as the large problem is copied across in small blocks and an overhead in reordering the block list ($O(N\log_2 P)$). While memory scalability is not seriously compromised, such a system cannot scale well computationally without using a parallel file system. Data striping to parallel files simply requires that as each processor prepares to send a block, the block is filled only with data that falls into the range requested by the i/o processor.

3.3 Scattering data from file

Many applications require that, in addition to the mesh, some global data is required from file. This can be handled in a similar manner to the scalable gather.

Small blocks of memory on the i/o processor(s), one for each processor, hold the global entity numbers for the block, received from the appropriate processor. A current position is maintained for each block and a block list is ordered according the global entity numbers in each block's current position. Data read from file is copied into the current block at the current access position, looping for consecutive global entity numbers within each block and re-ordering the block list when switching to a new block. Each block is again maintained in two halves so that as a block becomes half full it is sent to the appropriate processor which returns the next (half) block of global entity numbers. Again this allows the communication of data from and to the i/o processor to be concurrent with file input operation.

3.4 Localised graph partition optimisation

To reduce the bottleneck of mesh decomposition and distribution, the run-time calculation of data dependencies across an unstructured mesh and the calculation of a mesh partition must both be localised. That is, any data originating from a file system must be distributed, as rapidly as possible, across the available processors, in

a manner that is at the absolute most $O(N)$. All subsequent operations on the data must be restricted to be internal to each data set (processor) or, at most, interactions between neighbouring data sets (processors). An unstructured mesh partition, however crude, possesses a sub-domain interconnection topology. In the interest of computational efficiency a mesh partition should reduce the degree of the sub-domain interconnection topology, and if possible, provide an efficient mapping of the partition topology onto the processor topology [3]. Localised optimisation algorithms in the graph partitioning code JOSTLE [7] has made it possible to localise the entire mesh decomposition process.

The algorithms in JOSTLE are designed to address the three problems that arise when partitioning unstructured meshes:

- The static partitioning problem (classical).
- The static load balancing problem, in particular for meshes generated in parallel.
- The dynamic load balancing/partitioning problem, as discussed in Sections 3.5.

In the second two cases, the initial data is a distributed local graph which may be inadequately partitioned. One way of dealing with this is to send the graph to some host processor, run a serial static partioning algorithm on it and then redistribute. Clearly this is unattractive for several reasons. An $O(N)$ overhead for mesh partitioning will not scale if the rest of the code is running at $O(N/P)$. Also, the graph may not fit into one processors memory. In addition, a partition of the graph, which may be perfectly adequate, already exists and may be used as a starting point for repartitioning [8]. For Dynamic Load Balancing (DLB) this is highly significant as, not only is the calculation of a new partition computationally expensive but, if this information is not used, then large quantities of data may be unnecessarily migrated without any reference to the data's current location. Therefore, because the graph is already distributed it is logical to repartition it *in situ*.

JOSTLE provides a highly localised optimisation algorithm with graph reduction to both accelerate the optimisation and, perhaps more importantly, provide a more global perception in a manner analogous to multigrid techniques. Most significantly for this work, JOSTLE iteratively optimises and, if necessary, load balances an existing partition in parallel.

P	$t_s(s)$	$t_p(s)$	speed up
16	1.59	0.18	8.83
32	1.97	0.19	10.37
64	2.61	0.18	14.50

Table 1: JOSTLE timings for a 224,000 node mesh using Cray T3E

JOSTLE makes it possible to read, from file, a mesh of N entities in, for example, blocks of N/P and crudely distribute the blocks across the processors. The block distribution defines the existing partition and local inspector loops create the distributed graph which is passed to JOSTLE. Communication of each block from the i/o processor to the destination processor may be concurrent with the often much slower file access. In which case the file access time for a given problem is approximately constant with increasing P ($O(N)$), which is not good, but allows subsequent operations on the mesh to become ($O(N/P)$). Without parallel file systems this bottleneck will persist, but it's impact can be reduced. The design of a similar initial distribution scheme to exploit a parallel file system is trivial.

3.5 Localised mesh decomposition

Consider the following example, a sequential code fragment where `NUMELE` is the global number of elements, `NUM_ADJ_ELE` is an array containing the number of elements adjacent to each element and `ELE_ELE` is an array containing the element to element connectivity (adjacency). The connectivity is represented as the number of each element adjacent to each face of each element. The two arrays `NUM_ADJ_ELE` and `ELE_ELE` that partially describe the mesh elements are read from file at run-time. While this example is necessarily simplistic the processes described are suitable for many more complex applications without serious difficulty.

```
INTEGER NUMELE, MAX_NUM_ADJ_ELE
INTEGER NUM_ADJ_ELE(1:NUMELE)
INTEGER ELE_ELE(1:MAX_NUM_ADJ_ELE,1:NUMELE)
```

```
DO I = 1, NUMELE
  NX(I) = 0
  DO J = 1, NUM_ADJ_ELE(I)
    NX(I) = NX(I) + X(ELE_ELE(J,I))
  END DO
  NX(I) = NX(I) / NUM_ADJ_ELE(I)
  X_MAX = MAX(X_MAX, NX(I))
END DO
```

With the two arrays read from file and distributed block-wise then, for NPROC processors, processor number PROCNUM = NPROC is allocated the first N/P elements. Processor number PROCNUM = NPROC-1 is allocated the next N/P and so on, leaving the last remaining elements on processor number 1 (i/o processor). So for all processors with the exception of processor number 1;

```
NUMELE = GLOB_NUMELE/NPROC
```

And for processor number 1;

```
NUMELE = (GLOB_NUMELE/NPROC) +
         MOD((GLOB_NUMELE,NPROC)
```

Where GLOB_NUMELE is the global number of elements.

The initial local mapping of element numbers from local to global numbering, held in the local array LOC_2_GLB_ELE, can be readily calculated from the initial block partition.

```
INTEGER LOC_2_GLB_ELE(1:NUMELE)
OFFSET = (NPROC - PROCNUM) * NUMELE
DO I = 1, NUMELE
  LOC_2_GLB_ELE(I) = OFFSET + I
END DO
```

A local inspector loop based on the example code loop is used to insert element pairs (graph edges) into the element to element graph ELE_ELE_GRAPH, in which each node represents an element and each edge represents element to element connectivity.

```
INTEGER ELE_ELE_GRAPH(1:2,1:MAX_NUM_ADJ_ELE)
INTEGER NEXT
NEXT = 1
DO I = 1, NUMELE
  DO J = 1, NUM_ADJ_ELE(I)
    ELE_ELE_GRAPH(NEXT) = LOC_2_GLB_ELE(I)
    ELE_ELE_GRAPH(NEXT+1) = ELE_ELE(J,I)
    NEXT = NEXT + 2
  END DO
END DO
```

This local inspector loop simply copies the two global element numbers LOC_2_GLB_ELE(I) and ELE_ELE(J,I) into the local element to element connectivity (adjacency) graph ELE_ELE_GRAPH. As I runs from 1 to NUMELE, then the global element numbers LOC_2_GLB_ELE(I) must be within the sub-domain, but the global element numbers ELE_ELE(J,I) may be outside the sub-domain. At least one element in each pair is therefore in the sub-domain core and so the resulting graph represents what is commonly referred to as a single layer overlap.

Each processor can now pass their graph ELE_ELE_GRAPH to parallel JOSTLE, which returns to each processor an array OWNER_OF_ELE containing the processor numbers that are to own each of the elements currently held on the processor together with an array NEIGHBOURS which contains the neighbouring processor numbers. As the initial block partion is unlikely to have been appropriate, redistribution of the element based data structures in accordance with OWNER_OF_ELE will certainly require a great deal of communication. It is possible that little or no entities stay on the same processor and those that do will need to be relocated in accordance to the new local numbering.

So for each processor number that appears in OWNER_OF_ELE, the redistribution routine copies into a local buffer the components of NUM_ADJ_ELE and ELE_ELE (and any other element based data structures) that are to be sent to that processor. These buffers (one of which may be required to stay on the same processor) are sent to the appropriate destination processor. As each of the buffers containing locally required sections of the element based data structures arrive at a processor they are unpacked into local memory and the global element numbers for each core element recorded in LOC_2_GLB_ELE. When all buffers have arrived, LOC_2_GLB_ELE is sorted to imply a local numbering in a globally ascending order, and the contents of NUM_ADJ_ELE and ELE_ELE (and other structures) are rearranged accordingly.

Each processor now has the required local element data structures but the contents of some structures still contain global mesh entity numbers. Renumbering such pointer arrays requires a global to local element mapping. In order to

provide a global to local mapping, without using a directly addressed, and consequently globally dimensioned and non-scalable data structure, the global numbers for each core element are stored in either; a local array, which, as it's contents are in ascending order, can be rapidly accessed with a binary search, or if memory permits, a hash table of binary trees, which provides faster access.

A local inspector loop is now used to calculate the receive components of a communication set, `ELE_ELE_COMSET`, consisting of the element numbers required to satisfy inter-sub-domain element to element dependencies. As such the receive communication set(s) also define the elment overlap (Section 2.1). Here the routine `ADD_TO_COMSET` requires `GLB_2_LOC_ELE` to be able to search for the global element number `ELE_ELE(J,I)` in the sub-domain core. If it cannot be found then it is added to the receive set in `ELE_ELE_COMSET`, which is sorted to remove duplicates on leaving the loop.

```
 DO I = 1, NUMELE
   DO J = 1, NUM_ADJ_ELE(I)
   CALL ADD_TO_COMSET ( ELE_ELE(J,I),
+          GLB_2_LOC_ELE, ELE_ELE_COMSET )
   END DO
 END DO
```

This loop may be sufficient to describe the entire element overlap but for many practical codes the required overlap is often a conjunction of several communication sets each calculated from separate inspector loops.

At this stage each processor has calculated the global element numbers required in the overlap but still needs to know which processor owns those elements. In calculating the mesh partition JOSTLE has to evaluate the sub-domain connectivity (topology), part of which is returned in an array `NEIGHBOURS` containing the local neighbouring processor numbers. This is used to localise searches for the processor that owns each of the overlap elements. The receive set of global element numbers in `ELE_ELE_COMSET` is copied to each processor listed in `NEIGHBOURS`. Consequently each processor receives a communication set from each neighbour. Global element numbers in each received set that are found in `GLB_2_LOC_ELE` are core elements, and so these global element numbers are returned to the sending processor, and locally recorded, as local element numbers in `ELE_ELE_COMSET`, as these core elements are the send communication set for element to element dependencies required by the neighbouring processor which sent the communication set. Clearly the communication sets are matched between neighbouring processors, a send set on one processor is a receive set on it's neighbour.

When all receive sets have been returned from the neighbours, the element numbers in each set can be sorted. Each element in each set is allocated in turn the next free local element number, the global element number is recorded in `LOC_2_GLB_ELE(I)`. The local element number is inserted into `GLB_2_LOC_ELE` and then copied into the receive set in `ELE_ELE_COMSET`. The send and receive sets in `ELE_ELE_COMSET` are now locally numbered and consistent across all processors. The contents of `ELE_ELE` can now be locally renumbered using `GLB_2_LOC_ELE`. With the addition of some communications the example code can now run in parallel without requiring any other modification.

```
CALL OVERLAP_UPDATE(X, ELE_ELE_COMSET)
DO I = 1, NUMELE
  NX(I) = 0
  DO J = 1, NUM_ADJ_ELE(I)
    NX(I) = NX(I) + X(ELE_ELE(J,I))
  END DO
  NX(I) = NX(I) / NUM_ADJ_ELE(I)
  X_MAX = MAX(X_MAX, NX(I))
END DO
CALL GLOBAL_MAX(X_MAX)
```

Here the call to `OVERLAP_UPDATE` is requested by the index of `X` indirected by `ELE_ELE` to address values of `X` in the sub-domain overlap. The communication set `ELE_ELE_COMSET` for element to element dependencies ensures communication of the necessary data. The call to `GLOBAL_MAX` is required to return to each processor the global maximum of `X_MAX` .

4 Conclusion

Efficient use of DM hardware requires that each processor operates on a sub–domain that is as large as will fit into it's memory (consistent with

load balance). The memory overhead required for buffering i/o processes has led to the construction of DM parallel machines in which the i/o processor(s) are provided with more memory than the other processors. Nevertheless the memory required for mesh partitioning places a severe restriction on the scalability of problem size.

The scalable gather/scatter operations require the i/o processor to provide P buffers the size of which is determined by the available memory on that processor. With increasing P then for constant N/P the problem size N increases but the size of each buffer must decrease to accommodate more buffers into memory. This results in inefficient communications due to increased numbers of messages and smaller data packets. It is arguable that for large P parallel file systems will be required but such technologies remain uncommon and present compatibility difficulties.

Parallel JOSTLE has made it possible to distribute not only the memory requirement for mesh decomposition but also the processing. The efficiency of these operations in parallel is not ideal but represents a major improvement over what was previously a sequential operation.

The schemes presented in this paper effectively reduce the previously significant mesh decomposition and i/o related scalability restrictions. A near maximal sub–domain size can now be allocated to each processor without requiring excessive i/o memory. The primary benefit is that it becomes possible to operate with efficiency on problems that were previously too large to partition in one processor's memory. In addition the start–up and shut–down bottle necks have been greatly reduced to allow improved interaction with large problem cases. The use of inspector loops has provided methods that are each sufficiently generic to allow automation of the techniques in CAPTools.

5 Acknowledgements

The authors wish to thank the following organisations:
Fujitsu European Centre for Information Technology for funding support and access to their AP3000 and VX/4 facilities.
The UK EPSRC for funding support from under the PSTPA program (Grant No. GR/K40321).

References

[1] C. S. Ierotheou, S. P. Johnson, M. Cross, and P.F.Leggett. Computer aided parallelisation tools (CAPTools) - conceptual overview and performance on the parallelisation of structured mesh codes. *Parallel Computing*, 22:163–195, 1996.

[2] S. P. Johnson, C. S. Ierotheou, and M. Cross. Inspector loop determination to reduce communication overheads in unstructured mesh code parallelisation. Technical Report PPRG-98-003, Parallel Processing Research Group, University of Greenwich, 1998.

[3] K. McManus. *PhD Thesis: A strategy for mapping unstructured mesh computational mechanics programs onto distributed memory parallel architectures*. PhD thesis, University of Greenwich, 1996.

[4] K. McManus, M. Cross, and S. Johnson. Issues and strategies in the parallelisation of unstructured multiphysics codes. In *Proc. PDCCM'97*, 1997.

[5] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proc. Second Int. Conf. on Supercomputing*, July 1988.

[6] Guy Robinson and Richard Lonsdale. Fluid dynamics in parallel using an unstructured mesh. Internal report, UKAEA, April 1990.

[7] C. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load balancing for distributed memory parallel systems. In *Proc. PDCCM'97*, 1997.

[8] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for unstructured meshes. *J. Par. Dist. Comput.*, 1998. (in press).