Developing Parallel Finite Element Software Using MPI

P.K. Jimack and N. Touheed
Computational PDE Unit
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK

Abstract: This paper presents an introduction to writing parallel finite element programs using the Message Passing Interface (MPI) library. It is assumed that the reader has a working knowledge of the finite element method but has little or no prior experience with parallel computing. Hence the paper begins with a short review of some of the key concepts in parallel programming under a distributed memory paradigm before moving on to discuss the details of a simple parallel finite element algorithm. Implementation details of this algorithm are then considered with reference to the C programming language and a number of extensions and improvements are discussed.

1 Introduction

This paper is targeted at readers with little or no prior experience of parallel programming for solving practical problems in computational mechanics and follows on directly from our introduction to MPI (Message Passing Interface) which appears in the proceedings of the previous Euroconference in this series [7]. More extensive tutorials in the use of MPI may be found in [4] or [13], however all of the prerequisite knowledge required for this paper is contained in [7].

The remainder of this paper is divided into four further sections. Section 2 provides some background material on MPI, much of which also appears in [7]. In Section 3 an algorithmic overview of a typical parallel finite element scheme is given, based upon the decomposition of the finite element mesh into a number of sub-meshes. Section 4 then presents an implementation of this scheme using MPI with C. The final section of the main paper (Section 5) is used to discuss generalisations of this implementation, including the use of non-blocking communication, the solution of more general classes of problem and the use of alternative finite element trial and test spaces. In addition to these sections there are also two significant appendices. The first of these contains a full listing of the parallel code developed in Section 4 whilst the second considers the practical issue of partitioning a given finite element mesh in advance of applying the parallel finite element solver. Code is also listed for this important preprocessing step.

2 Background to MPI

MPI provides a standard set of subprogram definitions which allow parallel programs to be written using a distributed memory programming model. In this paradigm a unique subset of the available memory is associated directly with each of the parallel processes. Only the memory associated with a particular process may be accessed by it and so computations can only be performed by a process on data stored in its own subset of memory. In order to allow more than one process to perform computations on a given set of data copies of this data must be sent to any process which requires it (to be saved on that process's memory). This is referred to as "message passing".

Over 100 C and Fortran message passing subprograms are defined by the MPI standard which was first published in 1994 ([11]). This library is now supported by almost all parallel computer manufacturers and numerous public domain implementations also exist to allow clusters of workstations to be used as if they were a single parallel machine. Moreover, it is possible (and often extremely efficient) to install versions of MPI on most shared memory computers so as to program them using a distributed memory model. The main advantage that has resulted from the acceptance of MPI as a message passing standard over the past few years is that scientists and engineers are now able to develop *portable* parallel programs for the first time. Prior to the widespread use of MPI each manufacturer tended to provide their own message passing routines which meant that a program written for one manufacturer's computers could not be directly ported to a different manufacturer's machines. (It should be noted however that there have been forerunners to MPI which have also addressed this issue and that of using clusters of workstations as a "virtual" parallel computer (see [14] for example).)

Since the original publication of the MPI standard in 1994 ([11]) there have been a number of enhancements, and so this original version of the library is referred to as MPI-1.1 (which also includes a small number of clarifications and minor corrections to the original document [11]). There is now also an MPI-1.2 and an MPI-2. The former contains further clarifications and corrections to MPI-1.1 whilst the latter includes a number of significant extensions (see [12] for complete details of both MPI-1.2 and MPI-2). None of these extensions will be used in the programs outlined in this paper although they include: support for parallel i/o, dynamic processes and one-sided communication, as well as specific bindings for both Fortran90 and C++.

In fact this paper makes use of a very small subset of the MPI library and all of the subprograms used are fully described in [7]. The two basic communication primitives that we utilise are the global reduction operation and point-to-point communications.

Global reduction is performed using the MPI function MPI_Allreduce. This function has six parameters[1] as shown below.

```
MPI_Allreduce(Send, Recv, Length, Type, Op, Comm);
```

The precise definition of these parameters is as follows.

Send stores the starting address of the memory location of the local data to be reduced.

Recv stores the starting address of the memory location of the result of the reduction.

Length stores the number of data items in the Send memory location.

Type stores the single data type of each data item in the Send memory location (this will be one of MPI_CHAR, MPI_INT, MPI_FLOAT, etc. – see [7] for a full list).

Op stores the type of reduction operation to be performed and whose result is to be stored in the Recv memory location (this will be one of MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc. – see [7] for a full list).

Comm stores the name of the communicator to be used for the reduction (this is effectively the set of processes which will take part in the reduction operation and, for this paper, will always take the predefined value MPI_COMM_WORLD which represents the set of all processes which have been launched).

Hence, if we wish to evaluate the sum of a set of real numbers which are stored one per process and store a copy of this sum on the memory associated with each process we would use MPI_Allreduce with values of Length=1, Type=MPI_FLOAT, Op=MPI_SUM.

Point-to-point communication may be achieved using the pair of functions MPI_Send and MPI_Recv. These are defined as follows.

```
MPI_Send(Send, Length, Type, Dest, Tag, Comm);
```

and

```
MPI_Recv(Recv, Length, Type, Source, Tag, Comm, Status);
```

with MPI_Send being called by the process which is sending the message and MPI_Recv being called by the process which is to receive it. The precise definition of the parameters is as follows.

Send stores the starting address of the memory location on the sending process where the data to be sent is located.

[1] In the Fortran binding there is always an additional parameter which is used to return an error code. This error code is returned through the function value in the C binding. From now on we will only refer to the parameter lists for the C binding however the reader is referred to the appendix of [7] for a full discussion of the minor differences between the C and Fortran bindings.

Recv stores the starting address of the memory location on the receiving process where the received data is to be stored.

Length stores the number of data items to be sent/received.

Type stores the single data type of each of the data items to be sent/received.

Dest stores the rank (i.e. process number) of the process to which the message is to be sent.

Source stores the rank of the process from which the message should be received (this may take the predefined value MPI_ANY_RANK if the source process is unimportant).

Tag stores an integer tag which may be used to identify an individual message or a member of a particular set of messages (this may take the predefined value MPI_ANY_TAG on the receiving process if the tag of the message is unimportant).

Comm stores the name of the communicator (i.e. group of processes) from which the process ranks should be selected (again we always use the group MPI_COMM_WORLD in this paper).

Status is of the type MPI_STATUS and, on exit from MPI_Recv, contains information on the receiving process about the message that arrived: its source, its tag and an error code (the first two may be useful if MPI_ANY_SOURCE and/or MPI_ANY_TAG have been used).

These two functions execute sends and receives with blocking. This means that control will not return from these function until a copy of the data to be sent has successfully left the Send address or arrived at the Recv address respectively. When returning from a call to MPI_Send one cannot assume that a message has been successfully delivered: only that it has been buffered and is "on its way". It is safe to overwrite the memory location pointed to be Send however. The important issue of the buffering of messages to be sent will be returned to in section 5.

Before finishing this section on MPI, we briefly mention a few more essential features.

- All MPI programs must include the file *mpi.h* (or *mpif.h* for the Fortran binding[2]) which contains all of the function prototypes and predefined MPI types and constants that a program may require.

- The first MPI function that a program called must always be MPI_Init. This allows any set-up procedures to be carried out on a particular parallel machine or workstation cluster. (Note that the issue of *how* an MPI program is actually started is not addressed by the MPI standard: this will vary from one operating system and architecture to another.)

[2] Due to the lack of global definitions in Fortran every subprogram which uses any MPI subroutines must include this file when the Fortran binding is used.

- The last MPI function that a program calls should be MPI_Finalize. This cleans up the parallel state before exiting and no other MPI function may be called after it (including MPI_Init).

- The function MPI_Abort may be used to (attempt to) abort all of the processes currently running in a given process group. This is typically used when an unexpected event or result occurs on a process. Its exact behaviour however is only defined when MPI_COMM_WORLD is used as the process group (i.e. attempt to abort all of the parallel processes currently running).

- Two further MPI functions are MPI_Comm_size and MPI_Comm_rank. These return the number of processes running in a group and the rank of the particular calling process within a group respectively. Once more, in this paper, only make use of the group which consists of all processes: MPI_COMM_WORLD.

Having briefly reviewed some of the fundamental concepts of MPI that we require for this paper we are now in a position to move on to the discussion of a particular parallel finite element algorithm. This will take place over the next two sections, with the first concentrating on the overall approach and the second on some specific implementation details.

3 A Parallel Finite Element Algorithm

Let us consider the second order linear elliptic partial differential equation

$$-\underline{\nabla} \cdot (a(\underline{x})\underline{\nabla} u) = f(\underline{x}) \ , \quad \underline{x} \in \Omega \subset \Re^2 \ , \tag{1}$$

subject to the Dirichlet boundary conditions

$$u|_{\partial\Omega} = g(\underline{x}) \ . \tag{2}$$

We will assume that the domain Ω is a bounded polygonal region which has been discretised into a non-overlapping set of triangles $\mathcal{T}$, and we will seek a piecewise linear finite element approximation, v, to u on this triangulation. Hence

$$v = \sum_{i=1}^{N} v_i N_i(\underline{x}) \quad + \quad \sum_{i=N+1}^{N+M} v_i N_i(\underline{x}) \ ,$$

where $N_i(\underline{x})$ are the usual piecewise linear basis functions on $\mathcal{T}$ (which has N interior vertices and M on the boundary), and v satisfies

$$\int_{\Omega} a \underline{\nabla} v \cdot \underline{\nabla} N_j \ d\underline{x} = \int_{\Omega} f N_j \ d\underline{x} \tag{3}$$

for $j = 1, ..., N$ (see [8] for example, for a full description of this Galerkin finite element procedure). The set of equations (3) may be written in matrix notation as

$$K\underline{v} = \underline{b} \ , \tag{4}$$

where

$$
\begin{aligned}
\underline{v} &= (v_1, ..., v_N)^T \ , \\
K_{ji} &= \int_\Omega a \underline{\nabla} N_j \cdot \underline{\nabla} N_i \ d\underline{x} \ , \\
b_j &= \int_\Omega f N_j \ d\underline{x} \ - \sum_{i=N+1}^{N+M} v_i \int_\Omega a \underline{\nabla} N_j \cdot \underline{\nabla} N_i \ d\underline{x} \ .
\end{aligned}
$$

To solve this problem in parallel it is first necessary to form the system of equations (4) in parallel and then to solve this system, again in parallel. This may be achieved by creating a partition of the triangulation $\mathcal{T}$, $\{\mathcal{T}_1, ..., \mathcal{T}_p\}$ say, where

$$\bigcup_{i=1}^{p} \mathcal{T}_i = \mathcal{T} \tag{5}$$

and

$$\mathcal{T}_i \cap \mathcal{T}_j = \{\} \quad \text{when } i \neq j \ . \tag{6}$$

It is then possible to assemble the contributions to the matrix K and the vector $\underline{b}$ in (4) independently (and concurrently) on p different processes, $i = 1, ..., p$, with process i working only on $\mathcal{T}_i$. Once this has been completed, an iterative algorithm may be used to solve (4) in parallel: we will describe an implementation of the conjugate gradient algorithm (see [3, chapter 10] for example).

For reasons that will soon become clear it is necessary to partition $\mathcal{T}$ in such a way that $|\mathcal{T}_i| \approx |\mathcal{T}|/p$ (i.e. each process deals with an approximately equal number of elements[3]) and the number of vertices lying on the partition boundary is as small as possible. Figure 1 shows an example of this for a quite uniform triangulation $\mathcal{T}$ whilst Figure 2 shows an example on a less uniform triangulation. In each case $p = 16$.

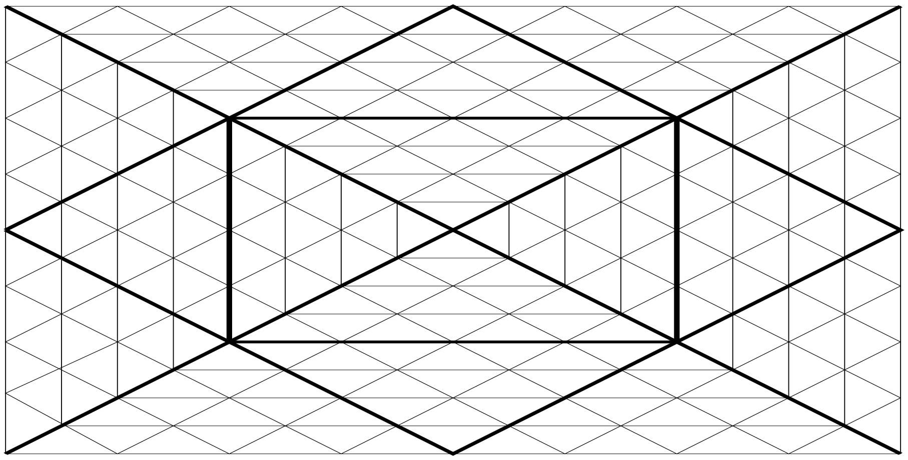

Figure 1: A partition of a uniform mesh into 16 pieces.

[3] We assume here that a homogeneous parallel computing system is being used in which each processor has the same performance characteristics.

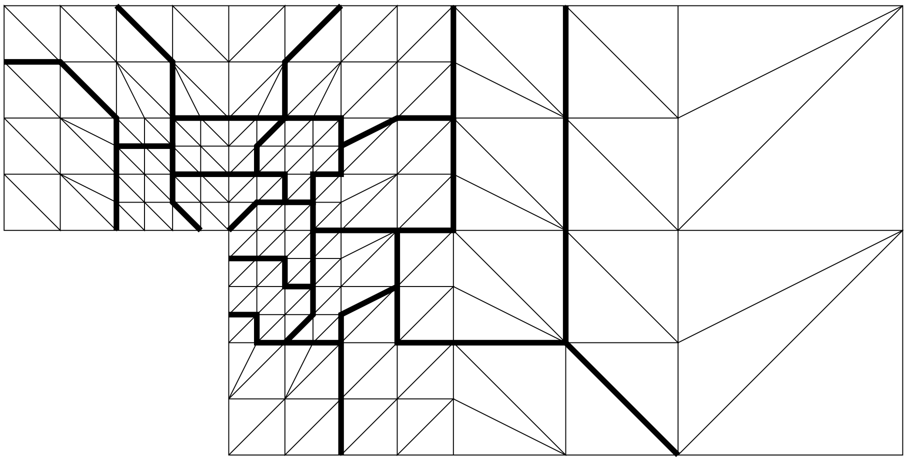

Figure 2: A partition of a non-uniform mesh into 16 pieces.

Suppose that, once the triangulation $\mathcal{T}$ has been partitioned, the unknowns, $\underline{v}$, are ordered in the following manner. Each of the interior vertices in $\mathcal{T}_1$ ($\underline{v}_1$ say), followed by each of the interior vertices in $\mathcal{T}_2$ ($\underline{v}_2$ say), etc., up to each of the interior vertices in $\mathcal{T}_p$ ($\underline{v}_p$), followed by each of the vertices lying on the partition boundary ($\underline{v}_s$ say). It then follows that the system (4) may be written in block matrix form as

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ B_1^T & B_2^T & \cdots & B_p^T & A_s \end{bmatrix} \begin{bmatrix} \underline{v}_1 \\ \underline{v}_2 \\ \vdots \\ \underline{v}_p \\ \underline{v}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_1 \\ \underline{b}_2 \\ \vdots \\ \underline{b}_p \\ \underline{b}_s \end{bmatrix} . \tag{7}$$

In (7) the block-arrowhead structure of the matrix K stems from the local support of the finite element basis functions. Moreover, each of the blocks A_i, B_i (and therefore B_i^T) and $\underline{b}_i$ may be computed entirely by process i (which works only with sub-mesh $\mathcal{T}_i$), for $i = 1, ..., p$. The blocks A_s and $\underline{b}_s$ both have contributions from elements in all of the sub-meshes however it is possible for each process to assemble the contributions to each of these only from those elements on its own sub-mesh. On process i we will refer to these contributions as A_{s_i} and $\underline{b}_{s_i}$ respectively.

Once each process has (concurrently) assembled each of the blocks A_i, B_i, $\underline{b}_i$, A_{s_i} and $\underline{b}_{s_i}$, the system (7) is stored in a distributed manner. It is then ready to be solved.

As indicated above, the solver that is considered here is an implementation of the conjugate gradient algorithm. In fact it is not a particularly efficient implementation since, for the sake of simplicity in this introduction, we do not make use of a number of key features. These include the use of a preconditioner (see [6], for example, for a discussion of parallel preconditioning which also contains a number of relevant references) or the use of static condensation (the Schur Complement method) to reduce the system (7) to one which involves only the interface unknowns, $\underline{v}_s$ (as described in [10] for example). In addition to this, the specific program described in the next section (and listed in appendix A) also avoids the use of sparse matrix data structures to maintain clarity (although there is significant exploitation of sparsity through the use of a local numbering convention in each process).

Figure 3 gives an outline of the parallel conjugate gradient algorithm that we use (taken from [5]) which is based directly on a typical sequential algorithm (see [3, chapter 10] for example). Inspection of this figure (which should be undertaken with reference to Figures 4 and 5 and Table 1) shows that each process i, for $i = 1, ..., p$, performs exactly the same algorithm and that the inter-processor communication is restricted to just two subprograms: InnerProd and Update.

```
1. $\underline{v}_i^0 = \underline{0}; \quad \underline{v}_{s_i}^0 = \underline{0}$
2. $\underline{b}_{s_i} = \text{Update}(\underline{b}_{s_i})$
3. $\underline{r}_i^0 = \underline{b}_i; \quad \underline{r}_{s_i}^0 = \underline{b}_{s_i}$
4. $\underline{p}_i^0 = \underline{b}_i; \quad \underline{p}_{s_i}^0 = \underline{b}_{s_i}$
5. $\gamma^0 = \text{InnerProd}(\underline{r}_i^0, \underline{r}_{s_i}^0; \underline{r}_i^0, \underline{r}_{s_i}^0)$
   For $k = 0, 1, ....$ Repeat steps 6-15.
      6.  $\underline{q}_i^k = A_i \underline{p}_i^k + B_i \underline{p}_{s_i}^k$
      7.  $\underline{q}_{s_i}^k = \text{Update}(B_i^T \underline{p}_i^k + A_{s_i} \underline{p}_{s_i}^k)$
      8.  $\tau^k = \text{InnerProd}(\underline{p}_i^k, \underline{p}_{s_i}^k; \underline{q}_i^k, \underline{q}_{s_i}^k)$
      9.  $\alpha^k = \gamma^k / \tau^k$
      10. $\underline{v}_i^{k+1} = \underline{v}_i^k + \alpha^k \underline{p}_i^k; \quad \underline{v}_{s_i}^{k+1} = \underline{v}_{s_i}^k + \alpha^k \underline{p}_{s_i}^k$
      11. $\underline{r}_i^{k+1} = \underline{r}_i^k - \alpha^k \underline{q}_i^k; \quad \underline{r}_{s_i}^{k+1} = \underline{r}_{s_i}^k - \alpha^k \underline{q}_{s_i}^k$
      12. $\gamma^{k+1} = \text{InnerProd}(\underline{r}_i^{k+1}, \underline{r}_{s_i}^{k+1}; \underline{r}_i^{k+1}, \underline{r}_{s_i}^{k+1})$
      13. if $(\sqrt{\gamma^{k+1}} \le \text{tol})$ END
      14. $\beta^k = \gamma^{k+1} / \gamma^k$
      15. $\underline{p}_i^{k+1} = \underline{r}_i^{k+1} + \beta^k \underline{p}_i^k; \quad \underline{p}_{s_i}^{k+1} = \underline{r}_{s_i}^{k+1} + \beta^k \underline{p}_{s_i}^k$
```

Figure 3: A parallel conjugate gradient algorithm to solve (7).

```
InnerProd($\underline{a}_1, \underline{a}_2; \underline{b}_1, \underline{b}_2$) {
   $\gamma = \underline{a}_1 \cdot \underline{b}_1 + \sum_i (\underline{a}_2[i] \times \underline{b}_2[i]) / (\text{Shared}[i])$
   AllReduce($\gamma$, sum, MPI_SUM)
return (sum) }
```

Figure 4: Algorithm for the function InnerProd in Figure 3.

The subprogram InnerProd is used to calculate the inner product of two distributed vectors in parallel and requires a single global communication. This communication is a global reduction which determines the sum of the contributions to the inner product from each process and then provides each process with a copy of this sum. Notice that for each vertex on the partition boundary the calculation of its contribution to the inner product is repeated on each process which contains this vertex as a boundary node. Hence these contributions are

```
Update($\underline{a}$) {
   for $i = 1$ to $p$
        for $j = 1$ to Common[$i$]
             Buf[$j$] $=$ $a$[Neighbour[$i, j$]]
        if Common[$i$] $> 0$
             Send(Buf, Common[$i$], $i$)
   for $i = 1$ to $p$
        if Common[$i$] $> 0$
             Receive(Buf, Common[$i$], $i$)
        for $j = 1$ to Common[$i$]
             $a$[Neighbour[$i, j$]]$+=$ Buf[$j$]
}
```

Figure 5: Algorithm for the function Update in Figure 3.

Shared[i] = number of processes which share the node on the sub-mesh boundary which has the local number i.
Common[i] = number of nodes shared between this process and process i (this is set to zero when $i =$ the rank of this process).
Neighbour[i,j] = local number of the node on the partition boundary which is the j^{th} node shared with process i.

Table 1: The data structures on each process (associated with Figures 4 and 5).

all scaled by the reciprocal of the number of processes which repeat the given calculation so as to ensure the correctness of the final result[4].

The subprogram Update is a little more complicated since it makes use of a number of point-to-point communications from each process to each of its neighbours[5]. The purpose of this subprogram is to allow distributed contributions to nodal values on the partition boundary to be assembled. Since, for a given node on the partition boundary, the contributions will only come from a small number of processes (see the partitions in Figures 1 and 2 for example) it would be inefficient not to perform these assemblies via local communication. Moreover, we do not wish each process to have to store the entire vector $\underline{v}_s$ (or $\underline{b}_s$, $\underline{r}_s$, $\underline{p}_s$, $\underline{q}_s$) but only the entries which correspond to vertices on its own part of the partition boundary. This may be achieved, and the communications required by Update may be successfully completed, provided that the local numbering of the partition boundary nodes on each sub-mesh is consistent. This means that any nodes on the boundary between two sub-meshes must be stored in the same

[4] This repetition of the calculation of some of the contributions to the inner product represents a parallel overhead which could be avoided by assigning an owning process to each of the vertices on the partition boundary and only performing the calculation on that particular process.

[5] Here we say that two processes are neighbours if the sub-meshes which they own have at least one common vertex on the partition boundary.

order on each of the processes to which the sub-meshes correspond. This may easily be ensured if the local numbering of the partition boundary nodes on each sub-mesh has the same ordering as some global numbering of the nodes in the entire mesh.

As a final clarification of the role of the subprogram Update it may be observed that the effect of step 2 of the algorithm in Figure 3 is to complete the assembly of the vector $\underline{b}_s$ of equations (7). Each process will then store those entries of $\underline{b}_s$ which correspond to the vertices on its own part of the partition boundary.

4 An Implementation Using MPI

The program listed in appendix A is a simple implementation of the parallel assembly and iterative solution procedures outlined in the previous section. It assumes that the finite element mesh has already been partitioned and is stored in a distributed manner in a number of files: one per sub-mesh (see appendix B for details of the partitioning process). Each process reads in its own sub-mesh from the file which is uniquely associated with it (via the process's rank in MPI_COMM_WORLD), and then assembles its own matrix blocks, Ap, Bp, Fp, As and Fs, corresponding to A_i, B_i, $\underline{b}_i$, A_{s_i} and $\underline{b}_{s_i}$ respectively in (7). The parallel conjugate gradient function (called CG) is then called.

The two main data structures used by this program, NodeType and ElementType, are for storing node and element information respectively. Each node has two coordinates which are stored in the Pos field of NodeType. There is also a field called Type which is used to indicate how many sub-meshes a node is shared between (a value of 0 indicates that the node is on the Dirichlet boundary, a value of 1 indicates that it is in the interior of the sub-mesh and a value of > 1 indicates that it is on the boundary of the sub-mesh). Finally, NodeType contains a field called Local which is used to allow separate local numberings for the nodes of Type 1 (from 0 to IntNodes $- 1$) and those of Type > 1 (from 0 to IBNodes $- 1$). The ElementType data structure contains just one field: Vertex, which is used to define the three vertices of an element.

The form of the input files containing the sub-meshes can vary significantly between different implementations. For this particular example we have chosen to first give the number of nodes and elements in the sub-mesh, followed by a list of the nodes (each represented by its two coordinates and its Type), then a list of the elements (represented by three vertices) and, finally, some information about which nodes are shared with which other sub-meshes. This information takes the form of one or two lines for each sub-mesh in the partition: the first line giving the total number of nodes shared with that sub-mesh, and the second giving a list of these nodes when the number is non-zero (note that the convention used is that zero nodes of the sub-mesh are shared with itself). As these different components of the mesh file are read in by each process, the data structures defined in Table 1 are also being assembled; as are the values of IntNodes and IBNodes. Finally, all of the arrays used by the program can now be allocated: including the array Buf which will be used for sending and receiving messages in the function Update. (Note that the calloc function has been used to allocate memory to those arrays which must have their entries initiated to zero.)

The assembly of the contents of arrays Ap, Bp, Fp, As and Fs is completed on each sub-mesh in essentially the same manner as for a sequential finite element code. For the particular implementation in appendix A it has been assumed that the vertices used to define each element are always listed in an anti-clockwise sense, that the function $a(\underline{x})$ in (1) is identically equal to 1, and that the function $g(\underline{x})$ in (2) is equal to x_1+x_2 (as defined by the function BC). Once the conjugate gradient function has been called all that remains is to output the solution in some way (this could easily be to a unique file rather than to standard output) and then call MPI_Finalize.

The conjugate gradient function, CG, follows the algorithm of Figure 3 very closely, as do the functions InnerProd and Update (which follow Figures 4 and 5 respectively). In the case of InnerProd use has been made of the MPI function MPI_Allreduce to perform the global communication, whereas the functions MPI_Send and MPI_Recv have been used for the point-to-point communications in Update (see Section 1).

5 Discussion

There are a number of features of the parallel finite element program in appendix A, described over the previous three sections, which are worthy of further discussion. Some of these, such as the use of static condensation to eliminate the interior unknowns for each sub-mesh ([10]) or the use of parallel preconditioners ([6]), have already been highlighted and are not considered further in this section. Others, such as the extension to nonlinear or non-self-adjoint differential equations or systems, will be discussed briefly here. The rest of this section will then focus on the function Update, with particular emphasis on the use of blocking or non-blocking communication routines.

5.1 Other Partial Differential Equations

The generalisation of the parallel finite element approach based upon domain decomposition to the solution of PDEs other than (1) is quite straightforward. Consider, for example, the linear non-self-adjoint modification of (1) given by

$$-\underline{\nabla} \cdot (a(\underline{x})\underline{\nabla} u) + \underline{b}(\underline{x}) \cdot \underline{\nabla} u = f(\underline{x}) \ , \quad \underline{x} \in \Omega \subset \Re^2 \ , \tag{8}$$

subject to similar boundary conditions as (2). The mesh $\mathcal{T}$ would be decomposed exactly as before and the finite element system derived using the same numbering for the unknowns, to yield a modified version of (7) which would not be symmetric:

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ C_1 & C_2 & \cdots & C_p & A_s \end{bmatrix} \begin{bmatrix} \underline{v}_1 \\ \underline{v}_2 \\ \vdots \\ \underline{v}_p \\ \underline{v}_s \end{bmatrix} = \begin{bmatrix} \underline{b}_1 \\ \underline{b}_2 \\ \vdots \\ \underline{b}_p \\ \underline{b}_s \end{bmatrix} . \tag{9}$$

Of course the entries of each of the blocks A_i, B_i, A_s, etc. would be different now since the entry in row j and column i of the stiffness matrix for equation (8) is given by

$$K_{ji} = \int_\Omega (a \underline{\nabla} N_j \cdot \underline{\nabla} N_i + N_j \underline{b} \cdot \underline{\nabla} N_i) \, d\underline{x} \, .$$

For such a non-symmetric system as (9) it is no longer possible to use the conjugate gradient algorithm, however other Krylov subspace methods, for example, my be applied instead (see [1] for example). As with the conjugate gradient solver the only communications that such parallel solvers would require at each iteration would be to evaluate inner products of distributed vectors and to update values of the shared nodes on the partition boundary.

Nonlinear PDEs may also be solved in parallel in a similar manner. For example, consider the equation

$$-\underline{\nabla} \cdot (F(\underline{u}) \underline{\nabla} u) + \underline{b}(\underline{x}) \cdot \underline{\nabla} u = f(\underline{x}) \, , \quad \underline{x} \in \Omega \subset \Re^2 \, , \tag{10}$$

again subject to the boundary conditions (2). For this problem the Galerkin finite element equations take the form of a nonlinear algebraic system, however the Jacobian of this system has exactly the same structure as the matrix in equation (9), and so the same parallel linear algebra solver may be used at each iteration of a Newton-like solver (see [2], for example, for a discussion on obtaining the Jacobian numerically in an efficient manner). Clearly the extension to systems of PDEs follows in a similar manner. The same algorithms may also be used for time-dependent problems by applying an implicit time-stepping strategy and solving a linear or nonlinear finite element system at each time-step.

5.2 Blocking Versus Non-Blocking Communication

The final point that we choose to discuss here, and possibly the most important, is the choice of point-to-point communications functions that have been used in the function Update. Both MPI_Send and MPI_Recv are examples of blocking communication routines. The precise way in which they work depends upon the implementation of MPI that is being used and upon the amount of memory that is available to the sending and receiving processes. In the case of MPI_Send for example, it is guaranteed that upon returning from a call to this function it is safe to overwrite the contents of the memory location Buf. This is because MPI_Send only returns once the contents of Buf has been forwarded, which either means that the message has been received by the destination process or, more usually, that a copy has been buffered and is in the process of being sent. If, when MPI_Send is called, there is no free memory on the sending process to buffer the message and the destination process is not yet ready to receive the message then the execution of the program is blocked at this point until it is possible to proceed.

Clearly, the way in which the function Update has been written is dependent upon each process being able to buffer each of the messages it is sending before going on to receive any messages. This means that there is a significant possibility of *deadlock* occurring when a very large problem is being solved (or if the implementation of MPI that is being used is inefficient at dealing with blocking sends). The problem of deadlock is one of the most important issues that must be addressed when writing parallel programs: if all of the processes are simultaneously blocking because they cannot proceed with sending a message then the whole parallel program comes to a halt and no further progress can be made.

In order to avoid this possibility of deadlock in Update it is necessary to make use of non-blocking communications functions. The non-blocking equivalent of MPI_Send is called MPI_Isend, which has one additional parameter to MPI_Send. This is a pointer of type MPI_Request and is used to return a unique identifier for the communication associated with each call to MPI_Isend (see [7], for example, for more details of this). The major difference that arises when using a non-blocking send is that, on exit from the function MPI_Isend it is *not* immediately safe to overwrite the data currently stored in the memory location Buf. Hence it is not sufficient to simply replace each call to MPI_Send in Update by a call to MPI_Isend, as this may lead to incorrect data being sent to a neighbouring process if Buf is overwritten before the message to a previous process was successfully sent. The simplest solution to this difficulty is to alter the declaration of Buf from

```
  float *Buf;
```

to

```
  float *Buf, *Buf2[PROCNO];
```

and then to malloc PROCNO extra arrays of type float, each of size Common[I] (for $0 \leq I <$ PROCNO). This would then allow Buf2[I,0] to locate the start of the message to be sent to process I and there would be no need to overwrite the contents of any memory locations before sending another message.

Note that by keeping the original one-dimensional array, Buf, for the blocking receives that are used in the second part of the Update function, there is never any possibility of either deadlock occurring or of overwriting memory locations too soon. The use of the existing blocking receives may not be particularly efficient however since some processes may spend a lot of time waiting for specific messages to arrive whilst they could be dealing with other messages that come first. Making careful use of MPI_ANY_SOURCE, or even non-blocking receives, could improve this situation significantly. We therefore finish this brief introduction by suggesting that a good exercise for the reader would be to modify the code given in appendix A in order to improve the reliability and efficiency of the function Update in the manner outlined here.

References

[1] S.F. Ashby, T.A. Manteuffel and P.E. Taylor, *"A Taxonomy for Conjugate Gradient Methods"*, SIAM J. Numer. Anal., 27, 1542–1568, 1990.

[2] C.J. Capon and P.K. Jimack, *"An Inexact Newton Method for Systems Arising from the Finite Element Method"*, Appl. Math. Letters, 10, 9–12, 1997.

[3] G.H. Golub and C.F. Van Loan, *"Matrix Computations"*, 2nd edition, John Hopkins University Press, 1989.

[4] W. Gropp, E. Lusk, A. Skjellum, *"Using MPI: Portable Parallel Programming with the Message-Passing Interface"*, MIT Press, Cambridge, Massachusetts, 1994.

[5] D.C. Hodgson and P.K. Jimack, *"Efficient Mesh Partitioning for Parallel Elliptic Differential Equation Solvers"*, Comput. Sys. in Eng., 6, 1–12, 1995.

[6] P.K. Jimack and D.C. Hodgson, *"Parallel Preconditioners Based Upon Domain Decomposition"*, In Parallel and Distributed Processing for Computational Mechanics I, ed. B.H.V. Topping (Saxe-Coburg Publications), 1997.

[7] P.K. Jimack and N. Touheed, *"An Introduction to MPI for Computational Mechanics"*, In Parallel and Distributed Processing for Computational Mechanics I, ed. B.H.V. Topping (Saxe-Coburg Publications), 1997.

[8] C. Johnson, *"Numerical Solution of Partial Differential Equations by The Finite Element Method"*, Cambridge University Press, 1987.

[9] G. Karypis and V. Kumar, *"A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.

[10] D.E. Keyes and W.E. Gropp, *"A Comparison of Some Domain Decomposition Techniques for Elliptic Partial Differential Equations and Their Parallel Implementation"*, SIAM J. Sci. Stat. Comput., 8, 167–202, 1987.

[11] Message Passing Interface Forum, *"MPI: A Message-Passing Interface Standard"*, International Journal of Supercomputer Applications, Vol. 8, No. 3/4, 1994.

[12] Message Passing Interface Forum, *"MPI-2: Extensions to the Message-Passing Interface"*, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, 1997.

[13] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *"MPI The Complete Reference"*, MIT Press, Cambridge, Massachusetts, 1996.

[14] V. Sunderam, *"PVM: A Framework for Parallel Distributed Computing"*, Concurrency: Practice and Experience, 2, 315–339, 1990.

[15] C. Walshaw, M. Cross and M.G. Everett, *"Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes"*, Proc. of 8th SIAM Conf. on Parallel Proc. for Scientific Computing, SIAM, 1997.

A Program Listing

In this appendix we provide a full listing of a parallel finite element program which is based upon the algorithm and implementation described in Sections 3 and 4 above. Improvements to, and generalisations of, this program are discussed in Section 5.

```
/* FemPar.c
   Written by Peter Jimack. */
/* A program using MPI (with C binding) compute a parallel
   finite element solution to a linear self-adjoint elliptic
   problem using conjugate gradient iterations. */

#include <stdio.h>
#include "mpi.h"

/* A number of global data structures and variables relating to
   the mesh and its partition are declared here. */

#define PROCNO 4
typedef struct
{
   float Pos[2];
   int Local;
   int Type;
} NodeType;
typedef struct
{
   int Vertex[3];
} ElementType;
NodeType *Node;
ElementType *Element;
int *Shared, MaxCommon, Common[PROCNO], *Neighbours[PROCNO];
int ProcNo, ProcID, Nodes, Elements, IntNodes, IBNodes;
float *Buf;

/* The following function calculates the inner product of two
   vectors distributed in a prescribed manner. */

float InnerProduct(float *A1, float *A2, float *B1, float *B2)
{
   int I;
   float IP = 0.0, IPTotal;
   for (I=0; I<IntNodes; I++)
      IP += A1[I] * B1[I];
   for (I=0; I<IBNodes; I++)
      IP += (A2[I] * B2[I]) / Shared[I];
   MPI_Allreduce(&IP, &IPTotal, 1, MPI_FLOAT, MPI_SUM,
                 MPI_COMM_WORLD);
   return(IPTotal);
}

/* The following function updates entries in the array A which are
   shared between more than one processor. */

void Update(float *A)
{
   int I, J;
   MPI_Status Status;
   for (I=0; I<PROCNO; I++)
   {
      for (J=0; J<Common[I]; J++)
         Buf[J] = A[Node[Neighbours[I][J]].Local];
      if (Common[I]>0)
         MPI_Send(&Buf[0], Common[I], MPI_FLOAT, I, 10,
                  MPI_COMM_WORLD);
   }
   for  (I=0; I<PROCNO; I++)
   {
      if (Common[I]>0)
         MPI_Recv(&Buf[0], Common[I], MPI_FLOAT, I, 10,
                  MPI_COMM_WORLD, &Status);
      for (J=0; J<Common[I]; J++)
         A[Node[Neighbours[I][J]].Local] += Buf[J];
   }
}

/* The following function implements the conjugate gradient
   algorithm in a distributed manner. */

int CG(float **Ap, float **As, float **Bp, float *Fp, float *Fs,
       float *Vp, float *Vs)
{
   float *Rp, *Rs, *Pp, *Ps, *Qp, *Qs, GammaOld, GammaNew, Tau,
      Alpha, Beta, Tol = 1.0e-4;
   int I, J, K, KMax = 250;

   Pp = (float*) malloc(IntNodes*sizeof(float));
   Ps = (float*) malloc(IBNodes*sizeof(float));
   Qp = (float*) malloc(IntNodes*sizeof(float));
   Qs = (float*) malloc(IBNodes*sizeof(float));
   Rp = (float*) malloc(IntNodes*sizeof(float));
   Rs = (float*) malloc(IBNodes*sizeof(float));
```

```
Update(Fs);
for (I=0; I<IntNodes; I++)
   Pp[I] = Rp[I] = Fp[I];
for (I=0; I<IBNodes; I++)
   Ps[I] = Rs[I] = Fs[I];
GammaNew = InnerProduct(Rp, Rs, Rp, Rs);
if (GammaNew < Tol*Tol)
   return(1);
if (ProcID == 0)
   printf("Gamma = %f\n",GammaNew);
for (K=0; K<KMax; K++)
{
   for (I=0; I<IntNodes; I++) {
      for (J=0, Qp[I]=0.0; J<IntNodes; J++)
         Qp[I] += Ap[I][J] * Pp[J];
      for (J=0; J<IBNodes; J++)
         Qp[I] += Bp[I][J] * Ps[J];
   }
   for (I=0; I<IBNodes; I++) {
      for (J=0, Qs[I]=0.0; J<IntNodes; J++)
         Qs[I] += Bp[J][I] * Pp[J];
      for (J=0; J<IBNodes; J++)
         Qs[I] += As[I][J] * Ps[J];
   }
   Update(Qs);
   Tau = InnerProduct(Pp, Ps, Qp, Qs);
   Alpha = GammaNew/Tau;
   for (I=0; I<IntNodes; I++) {
      Vp[I] += Alpha * Pp[I];
      Rp[I] -= Alpha * Qp[I];
   }
   for (I=0; I<IBNodes; I++) {
      Vs[I] += Alpha * Ps[I];
      Rs[I] -= Alpha * Qs[I];
   }
   GammaOld = GammaNew;
   GammaNew = InnerProduct(Rp, Rs, Rp, Rs);
   if (ProcID == 0)
      printf("Gamma = %f (K = %d)\n",GammaNew,K);
   if (GammaNew < Tol*Tol)
      return(1);
   Beta = GammaNew/GammaOld;
   for (I=0; I<IntNodes; I++)
      Pp[I] = Rp[I] + Beta*Pp[I];
   for (I=0; I<IBNodes; I++)
      Ps[I] = Rs[I] + Beta*Ps[I];
   }
   return(0);
}

/* The following function applies the Dirichlet boundary
   conditions. */

float BC(float X, float Y)
{
   return(X+Y);
}

/* The main function reads in the sub-mesh that is to be dealt
   with and then assembles the contributions to the stiffness
   matrix and load vector which come from the elements in this
   subdomain. The distributed conjugate gradient solver is then
   invoked. */

main(int argc, char** argv)
{
   FILE *InFile, *OutFile;
   char FileName[40];
   int I, J, K, II, JJ;
   float **Ap, **Bp, **As, *Fp, *Fs, *Vp, *Vs;
   float SLoc[3][2], TwoA, Area, Gr0[3], Gr1[3], StiffJI;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &ProcID);
   MPI_Comm_size(MPI_COMM_WORLD, &ProcNo);
   if (ProcNo!=PROCNO)
   {
      printf("Unexpected number of processors.\n");
      MPI_Abort(MPI_COMM_WORLD, -1);
   }
   if (ProcID < 10)
      sprintf(FileName, "Data0%d.In", ProcID);
   else
      sprintf(FileName, "Data%d.In", ProcID);
   InFile = fopen(FileName, "r");
   fscanf(InFile, "%d%d", &Nodes, &Elements);

   Node = (NodeType*) malloc(Nodes*sizeof(NodeType));
   for (I=0, IntNodes = 0, IBNodes = 0; I<Nodes; I++)
   {
      fscanf(InFile, "%f%f%d", &Node[I].Pos[0], &Node[I].Pos[1],
             &Node[I].Type);
   if (Node[I].Type == 1)
      Node[I].Local = IntNodes++;
   else if (Node[I].Type > 1)
      Node[I].Local = IBNodes++;
   else
      Node[I].Local = -1;
}
Shared = (int*) malloc(IBNodes*sizeof(int));
for (I=0, IBNodes = 0; I<Nodes; I++)
   if (Node[I].Type > 1)
      Shared[IBNodes++] = Node[I].Type;
Element = (ElementType*) malloc(Elements*sizeof(ElementType));
for (I=0; I<Elements; I++)
   fscanf(InFile, "%d%d%d", &Element[I].Vertex[0],
          &Element[I].Vertex[1], &Element[I].Vertex[2]);
MaxCommon = 0;
for (I=0; I<PROCNO; I++)
{
   fscanf(InFile, "%d", &Common[I]);
   if (Common[I]>MaxCommon)
      MaxCommon = Common[I];
   if (Common[I]>0)
      Neighbours[I] = (int*) malloc(Common[I]*sizeof(int));
   for (J=0; J<Common[I]; J++)
      fscanf(InFile, "%d", &Neighbours[I][J]);
}
Buf = (float*) malloc(MaxCommon*sizeof(float));
close(InFile);
printf("ProcID = %d, IntNodes = %d, IBNodes = %d\n",
       ProcID,IntNodes,IBNodes);
Ap = (float**) calloc(IntNodes, sizeof(float*));
for (I=0; I<IntNodes; I++)
   Ap[I] = (float*) calloc(IntNodes, sizeof(float));
Bp = (float**) calloc(IntNodes, sizeof(float*));
for (I=0; I<IntNodes; I++)
   Bp[I] = (float*) calloc(IBNodes, sizeof(float));
As = (float**) calloc(IBNodes, sizeof(float*));
for (I=0; I<IBNodes; I++)
   As[I] = (float*) calloc(IBNodes, sizeof(float));
Fp = (float*) calloc(IntNodes, sizeof(float));
Fs = (float*) calloc(IBNodes, sizeof(float));
Vp = (float*) calloc(IntNodes, sizeof(float));
Vs = (float*) calloc(IBNodes, sizeof(float));

for (K=0; K<Elements; K++) {
   SLoc[0][0] = Node[Element[K].Vertex[0]].Pos[0];
   SLoc[1][0] = Node[Element[K].Vertex[1]].Pos[0];
   SLoc[2][0] = Node[Element[K].Vertex[2]].Pos[0];
   SLoc[0][1] = Node[Element[K].Vertex[0]].Pos[1];
   SLoc[1][1] = Node[Element[K].Vertex[1]].Pos[1];
   SLoc[2][1] = Node[Element[K].Vertex[2]].Pos[1];
   TwoA = SLoc[0][0]*(SLoc[1][1]-SLoc[2][1]) +
          SLoc[1][0]*(SLoc[2][1]-SLoc[0][1]) +
          SLoc[2][0]*(SLoc[0][1]-SLoc[1][1]);
   Area = 0.5*TwoA;
   Gr0[0] = (SLoc[1][1]-SLoc[2][1])/TwoA;
   Gr0[1] = (SLoc[2][1]-SLoc[0][1])/TwoA;
   Gr0[2] = (SLoc[0][1]-SLoc[1][1])/TwoA;
   Gr1[0] = -(SLoc[1][0]-SLoc[2][0])/TwoA;
   Gr1[1] = -(SLoc[2][0]-SLoc[0][0])/TwoA;
   Gr1[2] = -(SLoc[0][0]-SLoc[1][0])/TwoA;
   for (J=0; J<3; J++) {
      JJ = Element[K].Vertex[J];
      if (Node[JJ].Type == 1)
         for (I=0; I<3; I++) {
            II = Element[K].Vertex[I];
            StiffJI = Area*(Gr0[I]*Gr0[J]+Gr1[I]*Gr1[J]);
            if (Node[II].Type == 1)
               Ap[Node[JJ].Local][Node[II].Local] += StiffJI;
            else if (Node[II].Type > 1)
               Bp[Node[JJ].Local][Node[II].Local] += StiffJI;
            else if (Node[II].Type == 0)
               Fp[Node[JJ].Local] -=
                  BC(Node[II].Pos[0],Node[II].Pos[1])*StiffJI;
         }
      else if (Node[JJ].Type > 1)
         for (I=0; I<3; I++) {
            II = Element[K].Vertex[I];
            StiffJI = Area*(Gr0[I]*Gr0[J]+Gr1[I]*Gr1[J]);
            if (Node[II].Type > 1)
               As[Node[JJ].Local][Node[II].Local] += StiffJI;
            else if (Node[II].Type == 0)
               Fs[Node[JJ].Local] -=
                  BC(Node[II].Pos[0],Node[II].Pos[1])*StiffJI;
         }
   }
}

CG(Ap, As, Bp, Fp, Fs, Vp, Vs);
for (I=0; I<Nodes; I++)
   if (Node[I].Type == 1)
      printf(" %f %f %f\n",Node[I].Pos[0],Node[I].Pos[1],
              Vp[Node[I].Local]);
   else if (Node[I].Type > 1)
      printf(" %f %f %f\n",Node[I].Pos[0],Node[I].Pos[1],
              Vs[Node[I].Local]);
MPI_Finalize();
}
```

B Mesh Partitioning

So far in this paper the issue of how to actually obtain a partition of a given finite element mesh has not been addressed. With the use of public domain graph partitioning tools this turns out to be a relatively straightforward issue to resolve. Two particularly good software tools are Metis ([9]) and Jostle([15]) which have been produced by researchers at the Universities of Minnesota[6] and Greenwich[7] respectively. These packages allow their users to partition an arbitrary graph into a number of subgraphs based upon fast multilevel algorithms (both sequential and parallel). In order to partition a finite element mesh it is first necessary to create the dual graph of this mesh. This is a graph which has a vertex for each element of the mesh and an edge joining two vertices whenever the elements to which these vertices correspond are adjacent. Metis provides a utility for creating such a dual graph from a finite element mesh automatically (the routine is called "mesh2dual"), hence this step is trivial. Now either Metis or Jostle may be used to partition this dual graph.

Both Metis and Jostle have exactly the same input and output formats so it is also trivial to switch between them. In particular, the output from each package takes the form of a file of numbers between 0 and $p-1$. Each number corresponds to a vertex of the dual graph (i.e. an element of the finite element mesh) and indicates to which subgraph (i.e. sub-mesh) it belongs. The following program, written in C, uses this information to create p new files which correspond to each sub-mesh in the partition. These files can be used as inputs to the parallel program listed in the previous appendix. The exact form of the input required by the program below is the number of elements followed by a list of elements (defined by their three vertex numbers), the number of nodes followed by a list of nodes (defined by their type ($< 0 \Rightarrow$ a Dirichlet boundary for example) and their coordinates), and then the output from Metis or Jostle.

```
/* Preprocess.c
   Written by Peter Jimack. */
/* A program in C to partition a single mesh into a number
   of distributed sub-meshes suitable for performing a parallel
   finite element solution. */

#include <stdio.h>
```

[6] http://www-users.cs.umn.edu/~karypis/metis/

[7] http://www.gre.ac.uk/~wc06/jostle/

```
/* A number of global data structures and variables relating to
   the mesh and its partition are declared here. */

#define FILENAME "Mesh.Data"
#define PROCNO 4
typedef struct
{
   float Pos[2];
   int GlobalDOF;
   int LocalNode;
   int Type;
} NodeType;
typedef struct
{
   int Vertex[3];
} ElementType;
NodeType *Node;
ElementType *Element;
int PartSize[PROCNO], *PartE[PROCNO], *PartN[PROCNO];

/* The main program reads in a mesh from a file followed by a
   list of which elements should be assigned to which
   subdomain. */

main(int argc, char** argv)
{
   FILE *InFile, *OutFile;
   int Nodes, Elements, I, J, K, V, NodeCounter[PROCNO],
      Common[PROCNO];
   char FileName[40];

   InFile = fopen(FILENAME, "r");
   fscanf(InFile, "%d", &Nodes);
   Node = (NodeType*) malloc(Nodes*sizeof(NodeType));
   for (I=0; I<PROCNO; I++)
      PartN[I] = (int*) malloc(Nodes*sizeof(int));
   for (I=0; I<Nodes; I++) {
      fscanf(InFile, "%d%f%f", &Node[I].GlobalDOF,
             &Node[I].Pos[0], &Node[I].Pos[1]);
      Node[I].Type = 0;
   }
   fscanf(InFile, "%d", &Elements);
   Element = (ElementType*) malloc(Elements*sizeof(ElementType));
   for (I=0; I<PROCNO; I++) {
      PartE[I] = (int*) malloc(Elements*sizeof(int));
      PartSize[I] = 0;
   }
   for (I=0; I<Elements; I++)
      fscanf(InFile, "%d%d%d", &Element[I].Vertex[0],
             &Element[I].Vertex[1], &Element[I].Vertex[2]);
   for (J=0; J<Elements; J++) {
      fscanf(InFile, "%d", &I);
      PartE[I][PartSize[I]++] = J;
   }
   fclose(InFile);
   for (I=0; I<PROCNO; I++) {
      NodeCounter[I] = 0;
      for (J=0; J<Nodes; J++) {
         Node[J].LocalNode = -1;
         PartN[I][J] = 0;
      }
      for (J=0; J<PartSize[I]; J++) {
         for (K=0; K<3; K++) {
            V = Element[PartE[I][J]].Vertex[K];
            if (Node[V].LocalNode == -1)
               Node[V].LocalNode = NodeCounter[I]++;
            if (Node[V].GlobalDOF > -1 && PartN[I][V] == 0) {
               Node[V].Type++;
               PartN[I][V] = 1;
            }
         }
      }
   }

/* For each processor a different file is created containing the
   sub- mesh that is to be stored on that processor. In addition
   to this sub-mesh each file contains lists of which nodes in
   that file are shared with each of the other processors. */

   for (I=0; I<PROCNO; I++) {
      if (I < 10)
         sprintf(FileName, "Data0%d.In", I);
      else
         sprintf(FileName, "Data%d.In", I);
      OutFile = fopen(FileName, "w");
      fprintf(OutFile," %d %d\n", NodeCounter[I], PartSize[I]);
      NodeCounter[I] = 0;
      for (J=0; J<Nodes; J++)
         Node[J].LocalNode = -1;
      for (J=0; J<PartSize[I]; J++) {
         for (K=0; K<3; K++) {
            V = Element[PartE[I][J]].Vertex[K];
            if (Node[V].LocalNode == -1) {
               Node[V].LocalNode = NodeCounter[I]++;
               fprintf(OutFile," %f %f %d\n", Node[V].Pos[0],
                      Node[V].Pos[1], Node[V].Type);
            }
         }
      }
      for (J=0; J<PartSize[I]; J++) {
         for (K=0; K<3; K++) {
            V = Element[PartE[I][J]].Vertex[K];
            fprintf(OutFile," %d",Node[V].LocalNode);
         }
         fprintf(OutFile,"\n");
      }
      for (K=0; K<PROCNO; K++) {
         Common[K] = 0;
         if (I!=K)
            for (J=0; J<Nodes; J++)
               if (PartN[I][J] > 0 && PartN[K][J] > 0)
                  Common[K]++;
      }
      for (K=0; K<PROCNO; K++) {
         fprintf(OutFile," %d\n",Common[K]);
         if (Common[K] > 0) {
            for (J=0; J<Nodes; J++)
               if (PartN[I][J] > 0 && PartN[K][J] > 0)
                  fprintf(OutFile," %d",Node[J].LocalNode);
            fprintf(OutFile,"\n");
         }
      }
      fclose(OutFile);
   }
}
```