Report no. 95/20

A parallel framework for unstructured grid solvers

D. A. Burgess P. I. Crumpton M. B. Giles

The aim of this work is the parallel solution of large two and three dimensional CFD problems using unstructured grids. A general *framework* has been formulated to enable both parallel and sequential execution of a single source FORTRAN code. This is achieved via the straightforward insertion of OPlus (Oxford Parallel Library for Unstructured Solvers) subroutine calls. Hence, the user's code can be developed, debugged and maintained on a sequential machine and executed in parallel when required. The generality is achieved through the specification by the programmer of certain key aspects of the data structure used in the computation. However, the programmer need not be aware of the underlying details of the parallel execution of the application and therefore the development of the parallel application is greatly simplified.

This work was presented at the ECCOMAS 1994 conference and appears in the proceedings (Computational Fluid Dynamics '94, Wiley, editors: S.Wagner, J.Periaux, E.H.Hirschel, pages 391–400).

This work was performed within Oxford Parallel. We gratefully acknowledge financial support from Rolls-Royce plc, DTI and SERC.

Oxford University Computing Laboratory
Numerical Analysis Group
Wolfson Building
Parks Road
Oxford, England OX1 3QD

April, 1997

1 Introduction

Algorithms for unstructured grids are becoming increasingly popular, especially within the CFD community where the geometrical flexibility of unstructured grids enables whole aircraft to be modelled. The resulting calculations are often huge and so there is a need to fully exploit modern parallel hardware.

Writing an individual, machine-specific parallel program is time consuming, expensive and difficult to maintain. Therefore there is a need for tools to simplify the task and generate very efficient parallel implementations. There have been several authors who have pursued the idea of constructing a library for parallelising programs for MIMD machines. These are mainly aimed at structured grids. De Keyser developed a software tool called LOCO [7] for structured grids with refined regions. Dellagiacoma *et al* constructed PARAGRID [5] which uses overlapping domain decomposition techniques on block–structured grids. Williams created DIME (Distributed Irregular Mesh Environment) [8] and Das *et al* developed PARTI (Parallel Automated Runtime Toolkit at ICASE) [3, 4] for unstructured mesh algorithms on distributed machines. However, DIME restricts the user to two dimensional triangular grid calculations. PARTI parallelises problems in any number of dimensions, and is the prior research which is closest in nature to the work reported here. There are a number of detailed differences between PARTI and OPlus, the library to be presented in this paper, but the principal difference is that with OPlus the programmer is not aware of the message-passing required for the parallel execution. This greatly simplifies the programmer's task. PARTI has the same objective but the aim is to achieve it through the incorporation of PARTI within an automatic parallelising compiler. At present, the programmer must still explicitly specify the message-passing to be performed.

The OPlus *framework* has been formulated to enable parallelisation of a large class of applications using unstructured grids. The present implementation is for distributed memory RISC machines. However, the user interface is designed to accommodate vector processors and/or shared memory machines. To utilise the library the application code needs to be written in a certain style which accommodates OPlus library calls. Thus a single source FORTRAN code can be developed, debugged and maintained on a sequential machine and then executed in parallel. This is of great benefit since parallelising unstructured grid codes is a time consuming and tedious process. Furthermore, considerable effort can be devoted to optimising the OPlus library for a variety of machines, using machine-specific low-level communications libraries as necessary. The performance benefits can then be realised by applications built using OPlus.

This paper will first describe the concepts behind the OPlus framework, then various aspects of the implementation and finally some results for a application code calculating the inviscid flow over an aircraft.

2 OPlus Library

The purpose of the OPlus library is to remove the parallelisation burden from the application programmer [1]. Emphasis is put on the following:

generality: OPlus uses general data structures. For example in a CFD application it allows cell, edge and/or face data structures. The cells can be of any type, such as triangles or quadrilaterals in 2D or tetrahedra, prisms or hexahedra in 3D.

performance: Messages are sent only when data has been modified, are concatenated to reduce latency, and communication is overlapped with computation whenever possible.

portability: OPlus is interfaced to general message-passing libraries such as PVM and MPI, or the native message passing library of the machine.

single source: A single source code can be executed either sequentially or in parallel. This greatly simplifies development and maintenance of parallel code.

2.1 Top level concepts

The concept behind the OPlus framework is that unstructured grid applications can be decomposed into four distinct parts.

sets Examples of sets are nodes, edges, triangular faces, quadrilateral faces, cells of a variety of types, far-field boundary nodes, wall boundary faces, etc.

data on sets Associated with these sets are *data*, for example the grid coordinates at nodes, the volumes of cells and the normals on faces.

pointers between sets In addition to data on sets there are pointers between sets, for example the cell to node connectivity which defines tetrahedra, the face to node connectivity which defines each face and the list of boundary nodes.

operations over sets All of the numerically-intensive parts of unstructured applications can be described as operations over sets. For example: looping over the set of cells using cell–node pointers and nodal data to calculate a residual and scatter an update to the nodes; or looping over the nonzeros of a sparse matrix accumulating a matrix-vector product.

The OPlus framework makes the important restriction that an operation over a set can be applied in *any* order without affecting the final result. Consequently, the OPlus routines can choose an ordering to achieve maximum parallel efficiency.

This restriction negates the use of OPlus for certain numerical algorithms such as Gauss–Seidel iteration or globally implicit ADI time-marching procedures. However, most numerical algorithms on unstructured grids in current use in CFD, and many other application areas, satisfy this restriction. Specific examples include explicit time-marching methods, multigrid calculations using explicit smoothing operators and conjugate gradient methods using local preconditioning.

Another restriction is that the sets and pointers must currently be declared at the start of the program execution and must then remain unaltered throughout the computation. Therefore, dynamic grid refinement cannot be treated at present. This is an area for future development.

2.2 Implementation Overview

The implementation uses a standard data-parallel approach. The computational domain is partitioned into a number of regions. Each partition is treated by a separate process, usually on a separate processor. A later section will discuss the handling of disk and terminal i/o through a master process.

Each member of a distributed set, and its associated data, is "owned" by a unique partition process. Other partitions may have temporary copies of the data, as needed. The key rule in performing parallel computations is that each partition performs all operations which cause modifications to data owned by it. Because of the restriction that operations can be performed in any order the different partitions can compute in parallel. At the partition interfaces, it is possible that an operation on a single set member (such as a cell) will affect data owned by more than one partition (such as data at the nodes). In this circumstance, the operation is performed on all of the affected partitions. This leads to redundant computations being performed. The alternative approach, adopted by PARTI, is to perform the computation on a single partition and subsequently communicate changes to the other partitions. This leads to more communication and a more complex programming interface.

To use the OPlus library the application programmer must

1. at the start of the code declare all the sets to be used within the code, through a simple OPlus subroutine call;

2. at the start of the code declare all the pointers to be used within the code, through another simple OPlus subroutine call;

3. adopt the OPlus loop syntax for loops over declared sets, see section 2.5;

4. use the OPlus i/o routines, see section 2.4.

The above is all that the application programmer need be concerned about. In particular, the programmer does not need to be aware of the partitioning of the sets performed by the OPlus library, or the message-passing that this generates.

2.3 Initialisation

At the beginning of the parallel execution there are three main initialisation phases:

partitioning All sets are partitioned and each partition is assigned an *owner* process. At present, the method adopted is to use a simple recursive inertial bisection algorithm to partition one or more sets. The other sets are partitioned consistently using the connectivity information contained in the pointers. It is also possible to read in partitions generated by other packages. This is a very active research area and general graph-based partitioning packages are becoming available.

construction of import/export lists If an operation on a member of a set requires data which is not locally owned then a copy of that data will need to be 'imported' from another partition. The initialisation phase constructs, for each partition, lists of the set members which may need to be imported during the main execution phase. Correspondingly, each partition also has export lists of the owned data which may need to be imported by other partitions.

local renumbering Each partition should only need to allocate sufficient memory to store the small fraction of each set which it either owns or imports. To enable this, it is necessary to locally renumber the set members. On each partition, each set is divided into a number of subgroups which are then numbered sequentially, and thus stored contiguously. The primary groups are the owned and imported data. Within these two groups there are further subgroups which depend on whether the set member needs to be executed (because its execution will affect owned data) and whether it requires imported data.

- **owned, interior** The member is owned and all of its pointers are to other members which are also owned. Thus, operations can be performed without reference to any imported data.

- **owned, halo** The member is owned, but at least one of the pointers is to imported data.

- **imported, executed** The member is imported, but at least one of its pointers is to an owned member so operations must be performed because it will affect the owned data.

- **imported, not executed** The member is imported, and none of the pointers are to owned members so there is no need for operations to be performed on this member.

The local renumbering of each set forces a consistent renumbering of all of the pointer information. The local-global mapping is also maintained for i/o purposes.

It is important to note again that all of the above phases are performed automatically by the OPlus library, not the application code. In all applications performed to date, the CPU time taken for these initialisation phases has been significantly less than the time required for the disk i/o, and so is considered to be negligible.

2.4 Input/Output

One of the important goals of the framework is to allow users to write a *single source code* which will execute either sequentially or in parallel depending on how the executable is linked. To achieve this it is necessary for the program to handle all disk and terminal i/o via appropriate subroutines.

1. For **sequential** execution the user's main program is linked to user–written subroutines which handle all i/o. This will enable the user to develop, debug and maintain their sequential code without any parallel message passing libraries.

2. For **parallel** execution the OPlus framework creates master and slave programs from the user's single source, Fig. 1. The master program is formed by linking the OPlus master process to the user's i/o routines, while the slave program is created by linking the user's compute process to OPlus slave routines.

Thus, when the slaves request data from a file on disk, the master process will read the global data and send each slave the data owned by the particular partition. Similarly for a slave write request, each slave will send the master its owned data, the master will concatenate and renumber the data and then call the user's subroutine to write it to the disk. Hence, i/o in this parallel environment is transparent to the user. This i/o is sequential in nature, but it is assumed that the parallel application will require very little i/o and so this will not become a bottleneck.

Although conceptually a master/slave paradigm is used, OPlus is implemented using a SPMD (single program multiple data) concept, to further simplify the interface to the user. Thus a Makefile produces two executables, one for sequential execution and one for parallel execution.

2.5 Loop Syntax

The following example is given to illustrate how a loop is parallelised. Suppose that for a set of triangular cells, the area of each cell, `AREAC`, is to be distributed to the cell's nodes using a pointer, NCELL, which points from the cell to its three nodes. This operation corresponds to the following FORTRAN DO–loop:

FORTRAN loop

```
DO IC = 1, NCELLS
  I1 = NCELL(1,IC)
  I2 = NCELL(2,IC)
  I3 = NCELL(3,IC)
  AREAN(I1) = AREAN(I1) + AREAC(IC)/3.0
  AREAN(I2) = AREAN(I2) + AREAC(IC)/3.0
  AREAN(I3) = AREAN(I3) + AREAC(IC)/3.0
ENDDO
```

Using the OPlus framework, this becomes:

FORTRAN OPlus loop

```
 DO WHILE(OP_PAR_LOOP(NCELLS,ISTART,IFINISH))
   CALL OP_ACCESS_R8('r',AREAC,1,NCELLS,NULL
&                    ,0,0,1,1)
   CALL OP_ACCESS_R8('u',AREAN,1,NNODES,NCELL
&                    ,1,1,1,3)
   DO IC = ISTART, IFINISH
     I1 = NCELL(1,IC)
     I2 = NCELL(2,IC)
     I3 = NCELL(3,IC)
     AREAN(I1) = AREAN(I1) + AREAC(IC)/3.0
     AREAN(I2) = AREAN(I2) + AREAC(IC)/3.0
     AREAN(I3) = AREAN(I3) + AREAC(IC)/3.0
   ENDDO
 END WHILE
```

The following comments discuss the transformation.

1. Firstly, note that there are no sends or receives in this parallel loop and the structure of the inner code remains unchanged. Thus all of the low level message-passing details are completely hidden from the user.

2. Essentially, the `DO WHILE` loop is similar to a colouring loop that would be necessary for vectorisation due to the inherent data dependencies. Indeed, future versions of the library for vector processors will automatically reorder the sets to enable a compiler directive asserting no data dependency to be safely inserted in front of the inner loop.

3. `OP_ACCESS` tells the library how the arrays in the main loop are to be accessed. Which distributed arrays need to be communicated is decided from the arguments of this routine. The first argument is a character string that can be legally set to `r,w,b,u` indicating read, write, both or update. This states how the second argument, the distributed array, is to be accessed in the loop. When the loop is executed sequentially the `OP_ACCESS` calls perform no function.

4. `OP_PAR_LOOP` is a logical function which returns as arguments the start and finish indices of the inner loop. In the sequential library `ISTART` and `IFINISH` are set to `1` and `NCELLS` respectively, to mimic the original FORTRAN loop. In the parallel library, this routine controls many passes through the inner loop:

   pass 1 analyse `OP_ACCESS` calls, fill a buffer with all data to be exported and bypass the inner loop by setting `ISTART=1` and `IFINISH=-1`.

   pass 2 Export all data in the buffer to neighbouring partitions, and perform the inner loop computation for those cells which do not require imported data.

   pass 3 Receive imported data from neighbouring partitions and perform computations for the remaining cells.

   At the conclusion of these three passes, all owned members of `AREAN` are correct, but imported values may not be.

Full details of the arguments and other routines can be found in [2].

3 Parallel Performance

To demonstrate the library a realistic industrial application has been chosen. This models the 3D steady inviscid flow past an aircraft, which is discretised using an unstructured tetrahedral grid, see Figure 2. A Lax–Wendroff pseudo timestepping algorithm is used as the solver. This can be expressed as

$$W_j := W_j + \omega_j N_j(W) \quad \forall \text{ nodes } j \tag{3.1}$$

where

$$N_j(W) := \frac{\sum_{\alpha \in C_j} V_\alpha \left[D_{\alpha,j} R_\alpha + A_{\alpha,j} \right]}{\sum_{\alpha \in C_j} V_\alpha}. \tag{3.2}$$

Table 1: Sets and pointers for Lax-Wendroff algorithm

set	size of set
nodes	137094
tetrahedral cells	746286
boundary faces	31536
boundary nodes	16123

pointer: from	to	length
tetrahedral cells	nodes	4
boundary faces	nodes	3
boundary nodes	nodes	1

Roman subscripts are used for nodal quantities, Greek subscripts for cell quantities and the other variables are:

ω_j is a relaxation factor.

C_j is the set of cells surrounding node j.

V_α is the cell volume.

R_α is the cell residual ($\oint F.nds$) corresponding to the inviscid Euler flux $F = (f, g, h)^T$.

$D_{\alpha,j}$ are distribution matrices mapping the cell residual R_α to node j.

$A_{\alpha,j}$ is an artificial dissipation term defined by

$$A_{\alpha,j} = \tau_4(\delta^2 W_0 - \delta^2 W_j) + \tau_2(W_0 - W_j)$$

where subscript 0 refers to a cell averaged quantity, τ_4, τ_2 are constants based on the timestep and $\delta^2 W_j$ is an undivided Laplacian operator.

There are two computationally intensive loops for each timestep:

1. a loop over cells, scattering $(W_0 - W_j)$ to nodes in order to construct $\delta^2 W$ at the nodes; this loop "reads" a W array and "updates" a $\delta^2 W$ array, so the read causes values of W to be imported at partition boundaries.

2. a loop over cells scattering $D_{\alpha,j} R_\alpha + A_{\alpha,j}$ to the nodes; this loop "reads" the W array, which is up-to-date from the previous loop, and the $\delta^2 W$ array, which is not up-to-date after the previous loop and so must be imported.

Table 2: Execution time per iteration and speed-up factor (S U) on IBM SP1 and SGI Challenge

		IBM(Enet)		IBM(switch)		SGI	
p	E	time	S U	time	S U	time	S U
1	0.0	23.4	1.0	23.4	1.0	45.3	1.00
2	2.0	12.3	1.9	12.3	1.9	23.0	1.97
3	3.8	9.3	2.5	9.0	2.5	15.9	2.85
4	5.4	7.6	3.1	7.1	3.3	12.4	3.65
5	6.6	7.1	3.3	6.4	3.6	—	—
6	7.9	7.5	3.1	5.5	4.2	—	—
7	8.0	7.7	3.0	5.0	4.7	—	—
8	9.7	8.0	2.9	4.5	5.2	—	—

There are also less important loops over boundary nodes to impose boundary conditions, and over nodes updating W to the next timestep.

The Lax-Wendroff algorithm requires four sets and three pointers, as given in Table 1. The surface grid and pressure contours for an airplane calculation are shown in Figure 2. The sizes of the four sets for this application are also given in Table 1. Execution times are given in Table 2 for the following two machines:

- an 8-processor distributed-memory IBM SP1,
- a 4-processor shared-memory SGI Challenge.

E is the percentage of redundant calculation being performed at partition interfaces, as described earlier. The elapsed times are per iteration; many thousands of iterations are actually required for numerical convergence. The columns labelled 'S U' give the speed-up relative to the single-processor execution. Two timings are quoted for the SP1, one using Ethernet for communication (Enet) the other using the switch. The Ethernet timings are equivalent to using a cluster of workstations. All timings use the PVM 3.2.6 software. Improvements in the switch and SGI timings are likely using native message passing routines and this is being investigated.

The percentage of redundant computation, increases with the number of processors, giving about 10% more work for eight processors. At first sight this might indicate a lack of scalability to a large number of processors. However, scalability for a fixed problem size is not the objective. The aim instead is to utilise parallel computers for large and complex industrial applications. The problem size for this example is only 0.75 Million cells; it is expected that future viscous calculations will have 10 Million cells. In this case the proportion of redundant calculations will remain small even when using more processors.

4 Visualisation

For such large applications it becomes prohibitive to rely on graphics workstations to manipulate and render the complete solution. To remedy this, the compute intensive part of the visualisation is performed on the parallel machine, along with the flow solver, while all the rendering is performed on the graphics workstation. This uses the pV3 software [6] which fits easily in the OPlus framework.

5 Conclusions

A flexible and general approach has been demonstrated to parallelise unstructured grid applications. This involves the programmer adopting the OPlus loop style of programming and all i/o being sent through specific subroutine calls. The resulting code will execute on a sequential machine (without the need for *any* parallel libraries) or in parallel on a MIMD architecture. This single source is of major benefit for development and maintenance of the code.

The OPlus parallel execution is fully optimised to concatenate messages, minimise the number of messages sent and overlap communication with computation. This library is intended for large applications, which warrant the use of parallel machines, and has been demonstrated by a 3D Euler solver for a complete aircraft configuration. For this realistic industrial application a worthwhile speed-up has been achieved with very little effort from the application programmer.

Acknowledgements

This work was performed within Oxford Parallel. We gratefully acknowledge financial support from Rolls-Royce plc, DTI and SERC.

References

[1] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A parallel framework for unstructured mesh solvers. IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Disributed Systems, 1994.

[2] P.I. Crumpton and M.B. Giles. OPlus programmer's manual. Oxford University computing laboratory, 1993.

[3] R. Das, J. Saltz, and H. Berryman. *A Manual for PARTI Runtime Primitives, Revision 1.* ICASE, NASA Lagley Research Centre, Hampton, USA, May 1993.

[4] R. Das, J. Saltz, J. Mavriplis, and R. Ponnusamy. The incremental schedular. In P. Mehrotra, J. Saltz, and R. Voigt, editors, *Unstructured scientific computation on scalable multiprocessors*, pages 81–105. MIT Press, Cambridge, MA, USA, 1992.

[5] F. Dellagiacoma, S. Paoletti, F. Poggi, and M. Vitaletti. PARAGRID: a parallel multi–block environment for Computational Fluid Dynamics. IBM ECSEC,Viale Oceano Pacifico 173, 00144 Rome, Italy.

[6] R. Haimes. pV3: A distributed system for large scale unsteady CFD visualisation. *AIAA Paper 94–0321*, 1994.

[7] J.De Keyser. *LOCO1.0: a library supporting data parallelism on MIMD computers.* Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, March 1993.

[8] R. D. Williams. *DIME Distributed Irregular Mesh Environment.* California Institute of Technology, 1990.

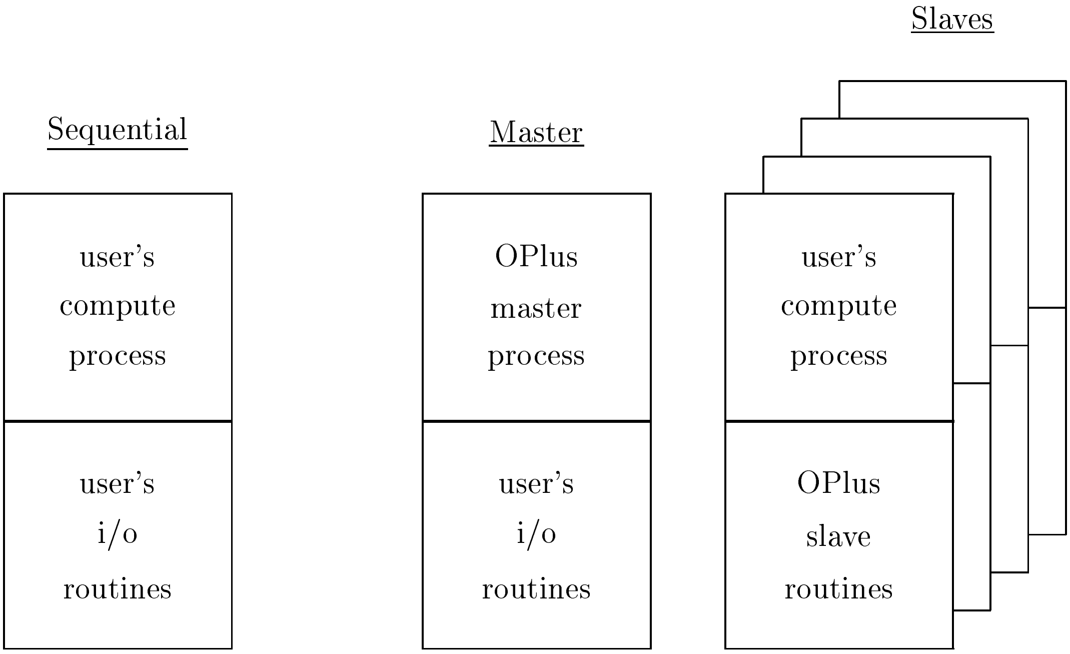

Figure 1: Sequential and parallel versions of user's program

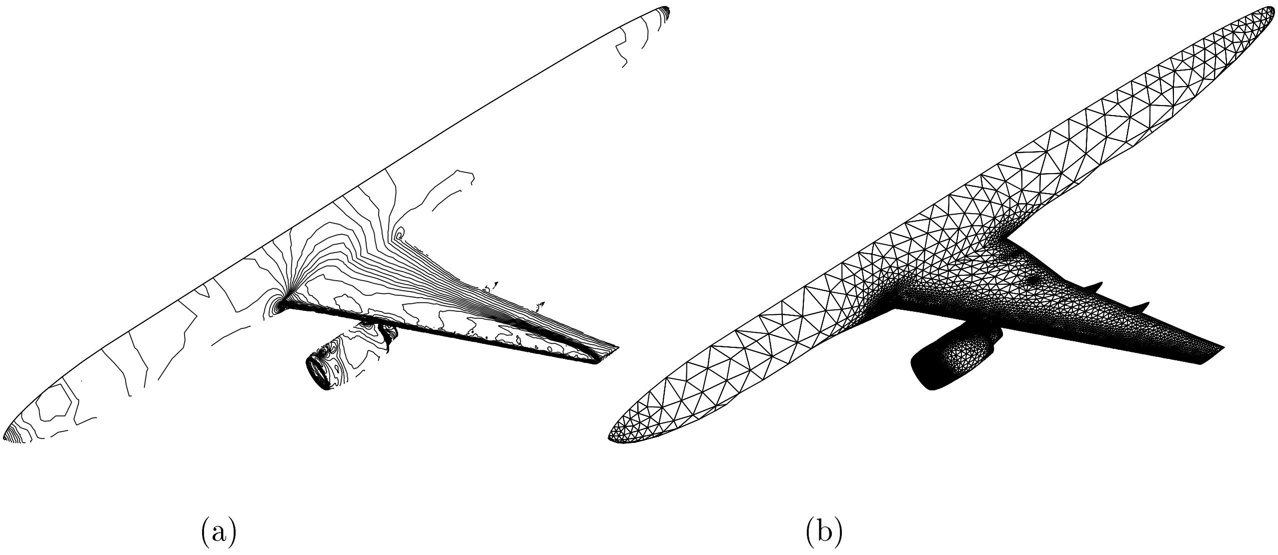

Figure 2: (a) contours of pressure, (b) surface grid