9th SIAM -PARALLEL PROCESSING CONFERENCE

9th SIAM PARALLEL PROC. CONF. 1999

Parallel Solution of Reacting Flow Problems using Unstructured Tetrahedral Meshes *

Martin Berzins† Paul M. Selwood ‡ Jonathan Nash §

Abstract

The use of parallel processing for the solution of air pollution problems makes it possible to solve problems in a fraction of the serial processor time. The parallel solution of reacting flow problems arising from simple models of atmospheric air pollution is described, using adaptive tetrahedral meshes in space and an explicit stiff chemistry time integration algorithm. The scalability and performance of the algorithm is considered and the performance of load balancing algorithms assessed. The difficulty of coding and debugging such applications is addressed by considering the use of high-level abstractions and shared abstract data types.

1 Introduction

The application of parallel computers to models of atmospheric air pollution makes it possible to generate solutions in a fraction of the normal time. In this paper an example code based upon adaptive tetrahedral meshes is considered and its parallel implementation described. This code is applied to an example problem consisting of a single plume arising from a power station with a simplified chemistry scheme involving seven species and a passive tracer, [10].

It is shown in [10] that the use of adaptive meshes in two dimensions can effectively increase resolution (and thence solution quality) while retaining acceptable solution times. In three dimensions adaptive mesh techniques can pick out the plume, but in order to obtain solutions with good spatial and temporal resolution serial computing times are prohibitive. The use of parallel adaptive meshes however gives a much faster solution time.

The parallel version of the code is written in a combination of ANSI C and the message passing standard MPI. The design of the code and its scalability for non-reacting flow problems are discussed in [7, 8]. The code is both robust and portable. A key issue is that of load-balancing the constantly evolving spatial mesh. Existing parallel load-balancing tools such as Jostle[12] and Metis[3] are used and an assessment made of their effectiveness for adaptive transient calculations. A key issue in the development of efficient scalable codes for such applications is the ease of writing portable programs. The paper concludes with a brief discussion of how high-level abstractions can be used with shared abstract data types to achieve this.

*Funded by EPSRC ROPA Grant GR/L73104

†Computational PDEs Unit, School of Computer Studies, The University of Leeds, Leeds, LS2 9JT, U.K.
‡Computational PDEs Unit, School of Computer Studies, The University of Leeds, Leeds, LS2 9JT, U.K.
§School of Computer Studies, The University of Leeds, Leeds, LS2 9JT, U.K.

2 Atmospheric Diffusion Equation Example

The application considered here is taken from a model of atmospheric dispersion from a power station plume - a concentrated source of NOx emissions, [10]. The photochemical reaction of this NOx with polluted air leads to the generation of ozone at large distances downwind from the source. An accurate description of the distribution of pollutant concentrations is needed over large spatial regions in order to compare with field measurement calculations. The present trend is to use models incorporating an ever larger number of reactions and chemical species in the atmospheric chemistry model. The complex chemical kinetics in the atmospheric model gives rise to abrupt and sudden changes in the concentration of the chemical species in both space and time. These changes must be matched by changes in the spatial mesh and the timesteps if high resolution is required, [10]. The difference in time-scale between the reaction of these species leads to stiff systems of equations which require implicit numerical solvers and special linear equations solvers [1]. The requirement is thus to use a parallel unstructured adaptive mesh code with the capability of handling stiff source terms.

The power plant plume application is modelled by the atmospheric diffusion equation in three space dimensions given by:

$$(1) \qquad \frac{\partial c_s}{\partial t} = -\frac{\partial u c_s}{\partial x} - \frac{\partial v c_s}{\partial y} - \frac{\partial w c_s}{\partial z} + D(\frac{\partial c_s}{\partial x}, \frac{\partial c_s}{\partial y}, \frac{\partial c_s}{\partial z}) + R_s + E_s - (\kappa_{1s} + \kappa_{2s})c_s,$$

where c_s is the concentration of the s'th compound, u,v and w, are wind velocities, K_x and K_y are diffusivity coefficients and κ_{1s} and κ_{2s} are dry and wet deposition velocities respectively. E_s describes the distribution of emission sources for s'th compound and R_s is the chemical reaction term which may contain nonlinear terms in c_s. $D()$ is the diffusion term, which is set to zero here. For n chemical species an n-dimensional set of partial differential equations (p.d.e's) is formed where each is coupled through the nonlinear chemical reaction terms.

The simple test case model covers a region of 300 x 500 km. and is a three-dimensional form of that used by [10] and hence represents the main features which would commonly be found in an atmospheric model including slow and fast nonlinear chemistry, concentrated source terms and advection. The chemical mechanism contains only 7 species but still represents the main features of a tropospheric mechanism, namely the competition of the fast inorganic reactions, [10], with the chemistry of volatile organic compounds (VOC's), which occurs on a much slower time-scale. This separation in time-scales generates stiffness in the resulting equations. The reaction rate constants, the photolysis rates and the background concentrations listed by [10] are used in the model. The power station is taken to be the only source of NOx and this source is treated by setting the concentration in the chimney as an internal boundary condition. In terms of the mesh generation this ensures that the initial grid will contain more elements close to the concentrated emission source. The concentration in the chimney corresponds to an emission rate of NOx of 400kghr^{-1} and only 10% of the NOx to be emitted as NO_2. We have assumed a constant wind speed of 5ms^{-1} in the x-direction with y and z components of one tenth of this value.

Computational runs are typically carried out over a period of up to 48 hours so that the diurnal variations can be observed. Figure 1 shows the plume developing with the adaptive mesh clustered around the developed portion of the solution. The visualisation was realised by running the parallel code in conjunction with the SCIRUN system at the University of Utah, see [2]. The main area of mesh refinement is along the plume edges close to the chimney. Using the adaptive mesh, we can clearly see the plume edges and can easily

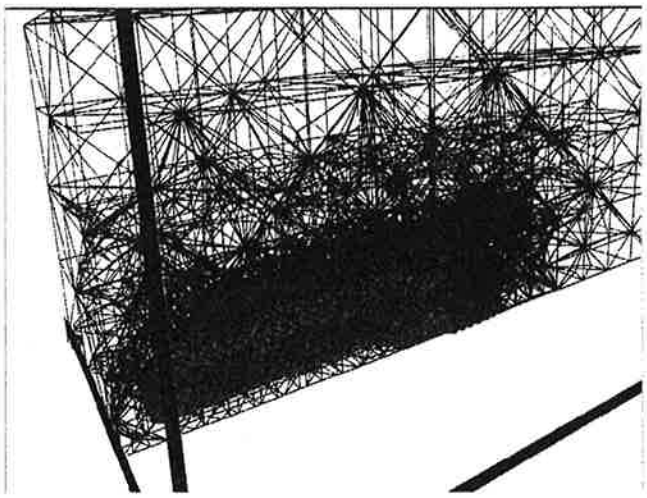


FIG. 1. *Example Tetrahedral Mesh for Reacting Flow problem.*

identify areas of high concentrations. The effects of the plume on ozone concentrations also provides some interesting results. Close to the plume the concentration of O_3 is much lower than that in the background. Due to the high NOx concentrations the inorganic chemistry is dominant in this region and ozone is consumed. As the plume travels downwind and the NOx levels decrease, the plume gradually picks up emissions of VOC's and leads to the production of NO_2 which pushes the above reaction in the reverse route. The levels of ozone can therefore rise above the background levels at quite large distances downwind from the source of NOx.

3 Numerical Solution Techniques.

Spatial discretization of the model atmospheric diffusion equation on unstructured tetrahedral meshes reduces the set of eight p.d.e's in four independent variables to a system of ordinary differential equations (o.d.e's) in one independent variable -time. This system of o.d.e's can then be solved as an initial value problem, [1]. For advection dominated problems it is important to choose a discretization scheme which preserves the physical range of the solution [9] and so a cell-centered, finite volume discretization is used in conjunction with spatial mesh adaptation based upon hierarchical refinement. The time integration approach adopted is that an implicit theta method is coupled to a spatial iteration in which a Gauss-Seidel scheme is used to solve the equations arising from the chemistry source terms [1].

Once the p.d.e's have been discretized in space we are left with a large system of coupled o.d.e's of dimension m $\times$ n where m is the number of mesh points, and n the number of species. These equations may now be written for a single species as

$$\dot{\underline{U}} \;=\; \underline{F}_N \;(\; t,\; \underline{U}(t)\;)\;,\; \underline{U}(0) \;\text{ given}\;, \tag{2}$$

where $\underline{U}(t) \;=\; [U(x_1, y_1, z_1, t), ..., U(x_N, y_N, z_N, t)\,]^T$. The point x_i, y_i, z_i is the centroid of the i th cell and $U_i(t)$ is a numerical approximation to the exact solution to the p.d.e. evaluated at the centroid i.e. $u(x_i, y_i, z_i, t)$. The time integrator computes an approximation, $\underline{V}(t)$, to the vector of exact p.d.e. solution values at the mesh points. This numerical solution at $t_{n+1} = t_n + k$, where k is the time step size, as denoted by $\underline{V}(t_{n+1})$,

is computed from

$$(3) \qquad \underline{V}(t_{n+1}) \;=\; \underline{V}(t_n) \;+\; (1-\theta)k\,\dot{\underline{V}}(t_n) + \theta\; k\;\underline{F}_N(t_{n+1}, \underline{V}(t_{n+1})),$$

in which $\underline{V}(t_n)$ and $\dot{\underline{V}}(t_n)$ are the numerical solution and its time derivative at the previous time t_n and $\theta = 0.55$. The equations to be solved for the correction to the solution $\underline{\Delta V}$ for the $p+1$ th iteration of the modified Newton iteration used with the Theta method are:

$$(4) \qquad [I - k\theta J]\;\underline{\Delta V} = \underline{r}\,(t^p_{n+1}) \quad where \;\; J = \frac{\partial\,\underline{F}_N}{\partial \underline{U}},\; \underline{\Delta V} = \left[\underline{V}(t^{p+1}_{n+1}) - \underline{V}(t^p_{n+1})\right]$$

and $\underline{r}\,(t^p_{n+1}) = -\underline{V}(t^p_{n+1}) + \underline{V}(t_n) + (1-\theta)k\dot{\underline{V}}(t_n) - \theta k\underline{F}_N(t_{n+1}, \underline{V}(t^p_{n+1}))$. The solution of this system of equations is the major computational task of the calculation. The cpu times are excessive unless special solution techniques such as splitting the nonlinear equations [1] into a set of flow terms and a reactive source term are employed. Consider the o.d.e. function $\underline{F}_N\;(t,\;\underline{U}(t)\;)$ defined by equation (4) and decompose it into two parts:

$$(5) \qquad \underline{F}_N\;(t,\;\underline{U}(t)\;) \;=\; \underline{F}^f_N\;(t,\;\underline{U}(t)\;) \;+\; \underline{F}^s_N\;(t,\;\underline{U}(t)\;)$$

where $\underline{F}^f_N\;(t,\;\underline{U}(t)\;)$ represents the discretization of the convective flux terms f and g in equation (1) and $\underline{F}^s_N\;(t,\;\underline{U}(t)\;)$ represents the discretization of the of the source term h in the same equation. The splitting approach used, [1], is to employ the iteration

$$(6) \qquad [I - k\theta J_s]\;\;\underline{\Delta V^*} \;\;=\;\; \underline{r}\,(t^p_{n+1})\,,\;\; J_s \;=\; \frac{\partial \underline{F}^s_N}{\partial \underline{U}},$$

where $\underline{\Delta V^*}$ is an approximation to $\underline{\Delta V}$. The advantage of this is that each block of equations corresponding to a tetrahedral element may be solved separately using the Gauss-Seidel based method of Verwer see [1]. The Jacobian matrix $[I - k\gamma J_s]$ is split into L, the strictly Lower triangular, D, the Diagonal and U the strictly Upper triangular matrices. and the equation rearranged to get

$$(7) \qquad (I - \gamma kD - \gamma kL)\Delta\underline{V}^*_{m+1} = \gamma kU\Delta\underline{V}^*_m + \underline{r}(t^p_{n+1}).$$

The matrix $I - k\theta J_s$ is the Jacobian of the discretization of the time derivatives and the chemistry source terms and is thus composed of independent diagonal blocks with as many block as there are tetrahedra. Each block has as many rows and columns as there are p.d.e.s. and each block's equations may be solved independently in parallel. Unlike operator splitting, this approach of splitting the nonlinear equations introduces no additional error, providing that the iteration converges, [1].

4 Mesh Generation and Adaptation

The initial mesh inside a rectangular bounding box was generated with either approximately 5000 or 38000 elements. This results in elements with side lengths of 10-50km For a power plant plume with a width of approximately 20km, it is impossible to resolve the fine structure within the plume using grids of this size,[10], hence our use of adaptive grids. Close to the chimney the mesh was refined to elements as small as 500m as this ensured that the mesh would be refined to a reasonable resolution in this region of steep gradients.

These meshes are then refined and coarsened by the parallel version (P)TETRAD [9] mesh adaptation module which is based on the refinement of tetrahedra into 8 tetrahedra

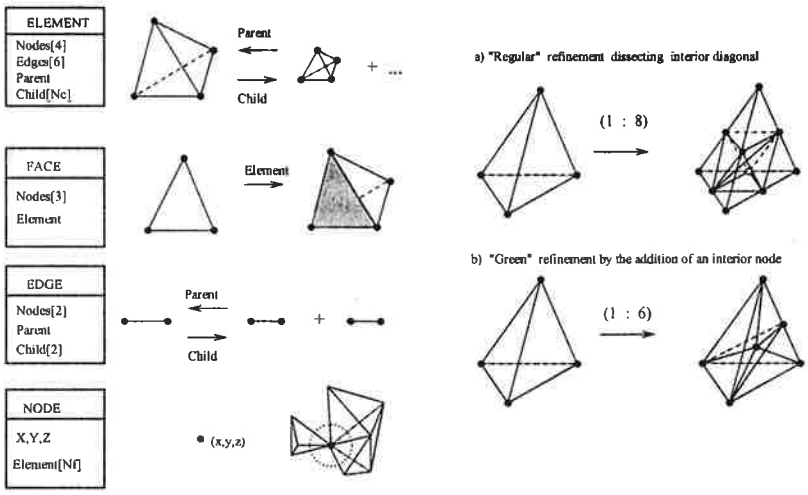


FIG. 2. *PTETRAD Data Structure and Refinement Examples.*

with appropriate adjustments to ensure that the mesh is conforming at the edges. Edges are first marked for refinement/de-refinement (or neither) according to some estimate or indicator. Elements with all edges marked for refinement are refined regularly into eight children. The remaining elements which have one or more edge to be refined use so-called "green" refinement. This places an extra node at the centroid of each element and is used to provide a link between regular elements of differing levels of refinement. The types of refinement are illustrated in Figure 2. Green elements are not refined further as this may adversely affect mesh quality, but are first removed and then uniform refinement is applied to the parent element.

Mesh de-refinement takes place immediately before the refinement of a mesh and only when all edges of all children of an element are marked for de-refinement and when none of the neighbours of an element to be deleted are green elements or have edges which have been marked for refinement. This restriction is to prevent the deleted elements immediately being generated again at the refinement stage which follows. TETRAD utilises a tree-based hierarchical mesh structure, with a rich interconnection between mesh objects. Figure 2 indicates the TETRAD mesh object structures in which the main connectivity information used is 'element to edge to node to element' and a complete mesh hierarchy is maintained by both element and edge trees.

The criterion for the application of the adaptivity used in this work is based on refining or coarsening the mesh based on the magnitude of solution gradients of the key chemical species NO and NO_2 across the faces of the tetrahedron, see [9]. For applications such as atmospheric modeling the maximum level of refinement is here limited to two or three levels to prevent excessive mesh adaptation close to point sources.

5 The Parallel Adaptation of Unstructured 3D Meshes

Parallel TETRAD was implemented using ANSI C with MPI due to the need for portability. The resulting code has been tested on a variety of platforms including Cray T3D and T3E, an SGI PowerChallenge, an SGI Origin 2000 and on a workstation network of SGI O2s. Communications are nearly always coalesced (to minimise latency and maximise bandwidth) and are performed by using nonblocking MPI functions to avoid deadlock and allow a degree of overlap with computation. The parallel computation of adaptive

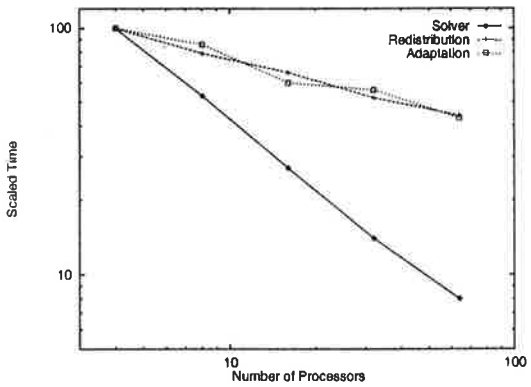


FIG. 3. *Scalability comparison of Solver, Adaptation and Redistribution.*

unstructured 3d meshes is typically based around distinct phases of execution, marked by the timesteps of the updated solution values, the mesh adaptation points, and the possible redistribution of the mesh elements. For PTETRAD, the grid is partitioned at the root or coarsest level, rather than at the leaf level, see [8]. Once the data is partitioned, the parallel version of the solver is straightforward to code due to the face data structure that exists within the adaptation software. The use of halo elements ensures that the owner of each face has a copy of all of the data (e.g. edges, nodes etc) required to complete the space discretization calculations provided the halo data is updated accordingly. These halo elements are used to reduce communications overheads, see [8]. In order to prevent data inconsistencies halo elements have one definitive owner.

5.1 Load Balancing

An equal work load for each processor is achieved initially through appropriately partitioning the original finite volume tetrahedral mesh across the processors then it is clear that the use of parallel adaptation for transient problems will cause the quality of the partition to deteriorate as the solution develops. Hence a parallel load-balancing technique is required which is capable of modifying an existing partition in a distributed manner. Parallel versions of Metis [3] and Jostle [12] are used to compute these new partitions; a comparison between these and a more recent algorithm for PTETRAD is given in [11]. All the algorithms produce partitions of similar quality.

5.2 Scalability

There are many factors that affect the scalability of the complete adaptive solution of a PDE. These include the number of remeshes required, the effectiveness of the remesh, the amount of work per adaptation step and the total depth of refinement. The solver effectively utilises explicit time integration and so scales well. Remeshing itself does not scale particularly well, and the costs associated with redistribution of the mesh can be quite high. An analysis of the scalability of the adaptivity is not straightforward but may be summarised by stating that relatively few processors are involved in refining elements and subsequently distributing relatively large amounts of data. Figure 3 gives a comparison of scaling for the final adaptation, repartitioning and computation phases in a gas dynamics calculation on a mesh with 34,560 base elements and 2 levels of refinement, see [8].

The solver clearly scales well, as would be expected for an explicit finite-volume scheme.

Case	P	Sol T.	Nremesh	Mesh	Halo	T.Steps
A	2	70.6	249	55.4	2.5	20.8
	4	35.4	274	58.3	6.6	23.0
	8	22.8	357	73.0	18.1	30.0
	16	8.0	298	83.2	30.1	25.1
B	8	17.5	73	249	31.8	5.8
	16	8.5	59	269	61.2	4.6
	32	7.0	73	384	146	5.7

TABLE 1
Cases A and B: Summary Statistics.

The adaptation and repartitioning scale at a similar rate to each other, but much more poorly than the solver. This is not surprising as they are far more communication intensive and do not have the work involved evenly distributed. It is interesting to note that in this case the repartitioning scales smoothly (this is not universally the case) while the adaptation is is rather oscillatory. This is likely to be a consequence of how well the partition happens to suit adaptation. In some cases (those with higher levels of refinement) the adaptation and repartitioning scale less well again due to the further increased communications and repetition of work involved in adapting the halos. Selwood and Berzins [8] also found that scalability improves as the amount of work per remesh is increased. The next step is to consider if this approach is suited to the atmospheric reacting flows described above.

5.3 Analysis of the Reacting Flow Calculation

In order to explore the performance of the reacting flow application two cases are considered. Case A is a two day run with a small initial mesh of 4800 elemnts while Case B is a half-day run with an initial coarse mesh of 38,400 elements. In both cases remeshing takes place after a fixed number of timesteps.

Time in the time in thousands of seconds, Mesh is the number of tetrahedra in thousands (including halo elements), Halo is the total number of halo tetrahedra in thousands, Sol.T, is the solver time for the next 80 timesteps(until the next remesh), Rem.T is the remesh time, Redist is the redistribution time, NmaxP is the maximum number of elements sent by a processor (in thousands of tetrahedra) in the redistribution phase.

Case A uses ParJostle, [12] with a coarsening threshold of 300. Good parallel speedup is seen for this case apart from some anomalous behaviour in the case of 8 processors. The speed-up did not however carry across to the 32 processor case. In order to investigate this a shorter time run with a larger initial mesh, Case B, was conducted. In this case the clear lack of speed-up between 16 and 32 processors is seen. An important indicator for this is the different numbers of timesteps and the different numbers of remeshings/redistributions (Nremesh) in each case. In order to investigate this further, Table 2 shows statistics for Case B, the larger initial mesh case, at five points in the integration. In this case redistribution is performed on the basis of information supplied by PMetis with the RepartMLRemap option. In Table 2 the maximum amount of data moved by any processor is shown by N_{maxP}. Many edges and nodes are also moved (in proportion to the number of elements moved). Both Selwood and Berzins, [8] and Oliker and Biswas [4] show that there is a reasonably good correspondence between the maximum number of elements that any processor has to move and the time taken for data redistribution (Redist). In contrast it is

Time	P	Mesh	Halo	Sol.T	Rem.T	Redist	NmaxP
5	8	92	15	123	4.7	3.5	7.9
	16	107	29	60	3.9	4.2	9.4
	32	152	69	48	4.8	3.1	7.0
10	8	155	25	231	7.9	9.4	20.2
	16	173	46	146	5.0	6.0	11.7
	32	260	117	89	6.0	3.7	10.1
15	8	192	28	298	6.2	12.1	31.5
	16	222	54	173	5.8	7.0	16.4
	32	351	157	116	8.0	4.2	9.3
20	8	245	31	254	8.6	14.2	28.9
	16	262	62	166	6.1	8.7	18.1
	32	405	171	107	8.2	4.2	4.7
25	8	247	33	253	10.7	-	-
	16	268	61	173	6.4	9.4	19.1
	32	390	153	100	7.2	4.5	6.0

TABLE 2
Case B: Statistics at Five Time Levels.

more difficult to generalise about the remeshing time. It is interesting to note that dynamic graph repartitioning algorithms minimise the total data moved rather than the maximum for a single processor, which this analysis suggests would be the more relevant metric.

One feature immediately evident from Table 2 is that the number of Halo elements has increased by a factor of three when moving from 16 to 32 processors. The time Sol.T shows the effect of this increase, but the total performance decrease is still not accounted for by this factor alone. Table 1 supplies the answer in that the code remeshes 59 times for 16 processors but 73 times for 32 processors. The solution is thus firstly to look at the PMetis partitions and also to debug the solver to find out why it is using 25 percent more timesteps and remeshes with 32 processors than with 16.

6 The Need for High-Level Programmming Abstractions

The debugging problem posed in the previous section is not attractive due to the low-level nature of the message passing code. The low-level implementation is not ideal for the complex irregular data-structures and large amounts of communication involved in parallel adaptation. This type of application is poorly supported by libraries and compilers however and message passing is the only real option. The main difficulty of working at the message passing level is that it is very difficult to maintain consistency between mesh objects and their copies, leading to tiresome and low-level debugging.

The need to improve the ease of programming such applications in parallel has led to the development of a two layered approach based on a high-level interface **SO**ftware **P**refetch **H**alo **I**nterface **A**bstraction or **SOPHIA** which allows the halo data to be prefetched when it is needed, [8]. This interface in turn makes use of **S**hared **A**bstract **D**ata **T**ypes. In contrast to abstract data types **ADT**s which support information sharing between different components of a serial application, **SADT**s can support sharing within an application executing across multiple processors. An SADT supports the clear distinction between

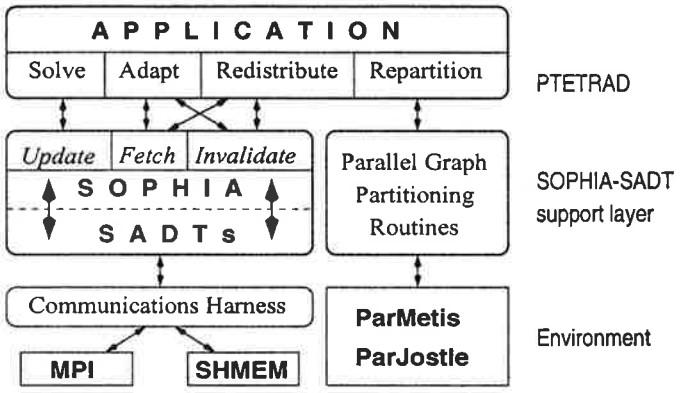


FIG. 4. *The new PTETRAD structure, Using SOPHIA and SADTs*

functionality and implementation, allowing its performance to be maximised on a given platform while preserving portability. The approach has led to the environment shown in Figure 4(b), with global mesh consistency supported by a set of SADTs, and local mesh access supported by a library of mesh routines. An SADT makes used of a small communications harness for data exchange, and alternative harnesses can be easily linked in on a given platform. Currently, MPI and the Cray/SGI SHMEM libraries are supported. The SOPHIA interface takes the form of the following primitives:

`SOPHIA_Fetch(local_data, shared_data)` This establishes a halo and its related communication patterns based on the distributed data and the required sharing. A full local copy of the required remote data is made to enable local computations to be made exactly as they would be in serial. In order for the fetch to be made, it is required that data on interprocess boundaries, together with off-processor connectivity, is specified at the initial partition stage. This connectivity is then stored either as a processor-pointer pair (for distributed memory machines) or just as a pointer (for shared memory) in order that the data structures under consideration may be traversed in order to complete the halo prefetch.

`SOPHIA_Update(shared_data, data_field)` This updates the given shared data with the current values of the specified data fields. By using knowledge of the application, only the necessary specified data fields are updated rather than the whole halo and thus communications can be minimised.

`SOPHIA_Invalidate(shared_data)` This removes a given halo from local memory. Careful use of invalidation followed by a new fetch enables e.g. changing the order of a solver partway through a CFD simulation.

This interface allows us to lift the abstraction above that of explicit messages passing, but with careful implementation, as shown below, the performance benefits of message passing should not be lost. It is particularly suited to applications, such as mesh adaptation, with irregular, complex data that varies significantly over time due to the ability to change the halos held by use of invalidation. Moreover the users knowledge of the application can be harnessed to ensure that halo updates are efficient.

An important SADT implements the SOPHIA function `SOPHIA_Update(shared_data, data_field)` in which the `data_field` part of halo information located on neighbouring mesh partitions must be maintained in a consistent state. Table 3 shows the performance of this SADT on mesh redistribution phase on 4 processors of the Origin 2000. The imbalance is reduced from 34% to 2%, with the repartitioning by Jostle taking around 1.15 secs in both cases. The redistribution of the mesh data is reduced by 10% in the new

	imbalance	repartition	redistribute	imbalance
PTETRAD	34%	1.15 secs	3.73 secs	2%
SADT version	34%	1.16 secs	3.37 secs	2%

TABLE 3

PTETRAD Original Version and Using SADTs

implementation, due to the subsequent tuning of both the local mesh access methods and the MPI communications strategies.

Acknowledgements

The authors would like to thank Alison Tomlin for providing the test problem and a long-standing collaboration,George Karypis and Chris Walshaw for supplying ever improving partitioning codes and Chris Johnson and the SCI Group for providing SCIrun, Fg 1. and access to their O2000.

References

[1] I.Ahmad I. and M.Berzins. *An Algorithm for ODEs from Atmospheric Dispersion Problems.* Appl. Num. Math. (25) 137-149 1997

[2] C.R. Johnson, M. Berzins, L. Zhukov, and R. Coffey. *SCIRun: Application to Atmospheric Dispersion Problems Using Unstructured Meshes.* pp. 111-122 in "Numerical Methods for Fluid Dynamics VI" (ed. M.J.Baines), ICFD, Wolfson Building, Parks Road, Oxford. ISBN 0 9524929 11, 1998.

[3] 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, (1997).

[4] L.Oliker and R. Biswas. *PLUM:Parallel Load Balancing for Adaptive Unstructured Meshes.* J. on Parallel and Distributed Computing, Sept/Oct. 1998.

[5] S.G. Parker, and D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. Modern Software Tools in Scientific Computing, Arge, E. and Bruaset, A.M. and Langtangen, H.P. editors, Birkhauser Press, 1-44, 1997.

[6] Jonathan Nash, Martin Berzins and Paul Selwood. A Structured Approach to the Support of a Parallel Adaptive 3D CFD Code. Submitted to proc EuroPar'99, Toulouse, France. 31 August - 3 September, 1999.

[7] P.M. Selwood, M. Berzins J. Nash and P.M. Dew *Portable Parallel Adaptation of Unstructured Tetrahedral Meshes.* pp 56-67 in "Solving Irregularly Structured Problems in Parallel" Proc. of Irregular 98 Conference (Ed. A.Ferreira et al.), Springer Lecture Notes in Computer Science, 1457, 1998.

[8] P.M. Selwood and M. Berzins *Parallel Unstructured Tetrahedral Mesh Adaptation: Algorithms, Implementation and Scalability* Submitted to Concurrency 1998.

[9] W. Speares and M. Berzins. *A 3D Unstructured Mesh Adaptation Algorithm for Time Dependent Shock Dominated Problems.* Int. Jour Num. Meths. in Fluids, 1997, 25, 81-104.

[10] A. Tomlin, M. Berzins J.M. Ware, J. Smith and M. Pilling. *On the use of adaptive gridding methods for modelling chemical transport from multi-scale sources* Atmospheric Env. Vol. 31 (18) 2945-2959.

[11] N. Touheed, P.M. Selwood, M. Berzins and P.K. Jimack, *"A Comparison of Some Dynamics Load Balancing Algorithms for a Parallel Adaptive Solver "*, Parallel and Distributed Processing for Computational Mechanics II (ed. B.H.V. Topping), Saxe-Coburg, 1998. ISBN 0-948 74954 7.

[12] C. Walshaw, M. Cross and M. Everett, *A Localised Algorithm for Optimising Unstructured Mesh Partitions*, Int. J. Supercomputing Appl, Vol. 9, No. 4, (1995).

Submitted to EUROPAR conference August 99

A Structured SADT Approach to the Support of a Parallel Adaptive 3D CFD Code

Jonathan Nash, Martin Berzins and Paul Selwood

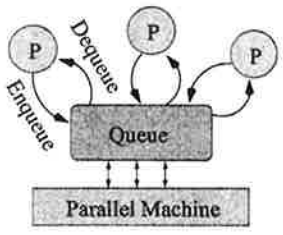
School of Computer Studies, The University of Leeds
Leeds LS2 9JT, West Yorkshire, UK

Abstract. The parallel implementation of unstructured adaptive tetrahedral meshes for the solution of transient flows requires many complex stages of communication. This is due to the irregular data sets and their dynamically changing distribution. This paper describes the use of Shared Abstract Data Types (SADTs) in the restructuring of such a code, called PTETRAD. SADTs are an extension of an ADT with the notion of concurrent access. The potential for increased performance and simplicity of code is demonstrated, while maintaining software portability. It is shown how SADTs can raise the programmer's level of abstraction away from the details of how data sharing is supported. Performance results are provided for the SGI Origin2000 and the Cray T3E machines.

1 Introduction

Parallel computing still suffers from a lack of structured support for the design and analysis of code for distributed memory applications. For example, the MPI library supports a portable set of routines, such that applications can be more readily moved between platforms. However, MPI requires the programmer to become involved in the detailed communication and synchronisation patterns which the application will generate. The resulting code is hard to maintain, and it is often difficult to determine which code segments might require further attention in order to improve performance. In addition, a portable interface does not imply portable performance - different MPI codes may have to be written to obtain good performance on new platforms.

Abstract Data Types (ADTs) have been used in serial computing to support modular and re-usable code. An example is a Queue, supporting a well-defined interface (Enqueue and Dequeue methods) which separates the functionality of the Queue from its internal implementation.



Whereas an ADT supports information sharing between the different components of an application, a Shared ADT (SADT) [6, 1, 3] can support sharing between applications executing across multiple processors. High performance in a parallel environment is supported by allowing the concurrent invocation of the SADT methods, where multiple Enqueue and Dequeue operations can be active across the processors.

The clear distinction between functionality and implementation leads to portable application code, and portable performance, since alternative SADT implementations can be examined without altering the application. The potential to generate re-usable SADTs means that greater degrees of investment, care and optimisation can be made in the implementation of an SADT on a given platform. For example, the implementation of a Queue on the Cray T3D [5] can support an increase in performance of 110, when the number of processors increase by a factor of 128. This was achieved by providing very high levels of fine-grain concurrency within the SADT implementation. In contrast, a more typical (lock-based) implementation has a performance ceiling of around 20.

In addition, an SADT can be parameterised by one or more user serial functions, in order to tailor the functionality of the SADT to that required by the application. For example, a shared Accumulator SADT [2] can produce a result based on the combined inputs supplied by each processor. A user function can be supplied which then determines the format of the inputs, and how those inputs are combined. The simplest case may be to sum an integer value at each processor. A more complex example is for each processor to submit a vector, and for the combining action to sum only the positive elements selected from each vector. This user parameterised form of the SADT is particularly useful in dealing with different parts of complex data structures in different ways.

This paper describes work [1] investigating the use of SADTs in a parallel computational fluid dynamics code, called PTETRAD [7, 8, 9]. The unstructured 3D tetrahedral mesh, which forms the basis for a finite volume analysis, is partitioned among the processors by PTETRAD. Mesh adaptivity is performed by recursively refining and de-refining mesh elements, resulting in a local tree data structure rooted at each of the original base elements. The initial mesh partitioning is carried out at this base element level, as is any repartitioning and redistribution of the mesh when load imbalance is detected.

A highly interconnected mesh data structure is used by PTETRAD, in order to support a wide variety of solvers and to reduce the complexity of using unstructured meshes. Nodes hold a one-way linked list of element pointers. Nodes and faces are stored as a two-way link list. Edges are held as a series of two-way linked lists (one per refinement level) with child and parent pointers. In addition, frequently used remote mesh objects are stored locally as halo copies, in order to reduce the communications overhead. The solver, adaptation and redistribution phases each require many different forms of communication within a parallel machine in order to support mesh consistency (of the solution values and the data structures), both of the local partition of the mesh and the halos. PTETRAD currently uses MPI to support this.

In this paper it will be shown how parts of PTETRAD may be used in SADTs based on top of MPI and SHMEM, instead of MPI directly, thus leading to software at a higher level of abstraction with a clear distinction between the serial and parallel parts of the code. Section 2 describes an SADT which has been designed to support the different mesh consistency protocols within an

[1] Funded by the EPSRC ROPA programme - Grant number GR/L73104

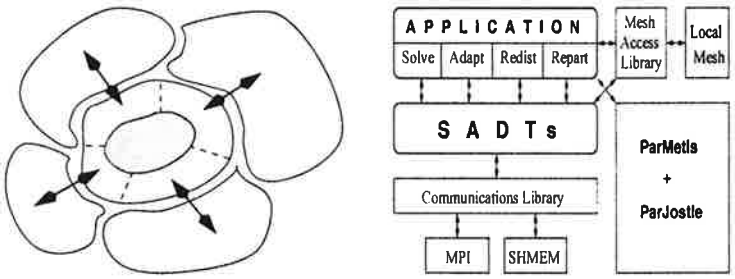


Fig. 1. (a) The SADT update method; (b) The new PTETRAD structure

unstructured tetrahedral mesh. A case study in Section 3 will describe the use of the SADT in supporting the mesh redistribution phase. A brief overview of the implementation techniques for the SADT will be given in Section 4, together with performance results for the SADT and for PTETRAD. The paper concludes by pointing to some current and future work.

2 An SADT for Maintaining Data Partition Consistency

The SADT described in this paper has focused on the problem depicted in Figure 1(a). A data set has been partitioned among p processors, with each processor holding internal data (the shared area), and overlapping data areas which must be maintained in a consistent state after being updated. PTETRAD maintains an array of pointers to base and leaf elements, which can be used to determine the appropriate information to be sent between partitions. For example, after mesh adaptation, any refined elements will require that their halo copies also be refined. Also, in the redistribution phase, the base element list can be used to determine which elements need to be moved between partitions. In constructing an SADT for this pattern of sharing, four basic phases of execution can be identified. The SADT contains a consistency protocol which specifies these phases of execution, with the generic form:

```
void Protocol (in, out)     /* Protocol interface with input...  */
{                           /* and output data lists.            */
  int send[p], recv[p];     /* Counters used in communications.  */
  pre-processing;           /* Initialisation of internal data.  */
  communications preamble;  /* Identifying how much data will... */
                            /* be exchanged between partitions.  */
  data communications;      /* Exchanging and processing the...  */
                            /* new data between the partitions.  */
  post-processing;          /* Format the results.               */
}
```

The protocol is called with a set of user-supplied input and output data lists (*in* and *out*), for example the lists of element pointers described above,

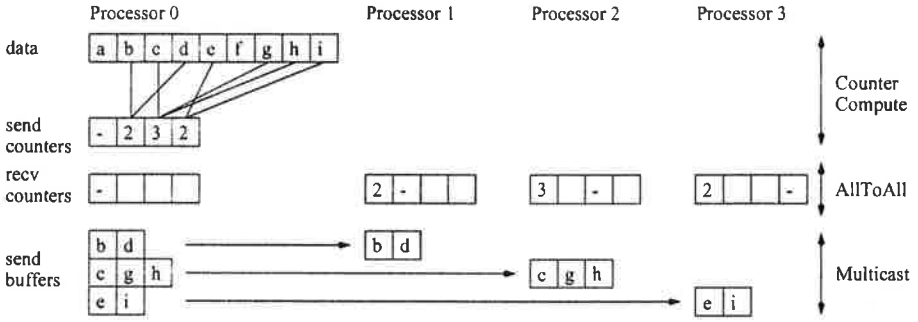


Fig. 2. The SADT communications stages

and each phase requires the user to supply a number of application-specific serial functions. The SADT is thus parameterised by these user functions which allow the communications phases to be tuned by evaluating various condition functions, for example to determine whether a data item is to be communicated to a given partition (if it has been refined or needs to be redistributed). The SADT also contains basic functions to pack/unpack selected fields of data items to/from message buffers, and to process the new data items which are received.

Figure 2 shows an example of the operation of the protocol. All processors will be performing the same phases of execution, but only the operation of processor 0 is shown here, for reasons of clarity (the pre-processing and post-processing phases are also removed). The protocol can make use of a communications library for global communications operations (denoted by `Comms_Function`), and may require one or more user-defined serial functions (denoted by `User_Function`).

(i) The communications preamble is given by:
- **(a)** `Comms_CounterCompute`: For each data item, `User_CounterCondition` decides if it is to be communicated, and `User_CounterIndexing` will update the associated values in the counters `send[]` and `recv[]`.
- **(b)** `Comms_AllToAll`: An all-to-all communication is executed, in which the other processors note the expected number of items to be received from processor i in `recv[i]` (if `Comms_CounterCompute` is able to determine the counter values in `recv[]` then this communication can be avoided).

(ii) The actual data communications phase is given by `Comms_Multicast`:
- **(a)** For each input item and each processor in turn, `User_SendCondition` decides if the item should be sent to the processor. `User_PackDatum` will choose the selected fields of the data item to send, and place them in a contiguous memory block, so that it can be copied into the message buffer for that processor (`User_DatumSize` allows the system to allocate the required total send and receive buffer space).
- **(b)** Once the buffers have been communicated between the processors, each item is removed in turn, using `User_UnpackDatum`, and the local data partition is updated, based on this item, with `User_ProcessDatum`.

3 Case Study: Mesh Redistribution

The new structure of PTETRAD [9] is shown in Figure 1(b). At the application level, local mesh access is supported by a library of mesh routines. The mesh repartitioning strategy is handled by linking in parallel versions of either the Metis or Jostle packages. The global mesh consistency is handled by making calls to the SADT, which also performs local mesh updates through the mesh library. The coordination between processors is supported by a small communications library which supports common traffic patterns, from a simple all-to-all exchange of integer values, up to the more complex packing, unpacking and processing of data buffers which are sent to all neighbouring mesh partitions.

The new SADT-based approach makes use of MPI, so that it may be run on both massively parallel machines and on networks of workstations, and also uses the Cray/SGI SHMEM library, to exploit the high performance direct memory access routines present on the SGI Origin 2000 and the Cray T3D/E. The use of an alternative communications mechanism is simply a matter of writing a new communications library (typically around 200 lines of code), and linking the compiled library into the main code.

The operation of the redistribution phase can be divided into four stages:

- **Repartition:** Compute the new mesh partitions at the base element level, using the parallel versions of Metis or Jostle.
- **Assign owners:** Update the local and halo owner fields for elements, edges, nodes and faces.
- **Redistribute:** Communicate the data to be moved and the new halo data.
- **Establish links:** Destroy any old communications links between local and halo mesh objects, and create the new links.

The following examples focus on the second stage of assigning the new owners for edges in the partitioned mesh, in which the halo edges must be updated with the new owner identifiers. PTETRAD maintains an array of edge lists, with each list holding the edges at a given level of refinement in the mesh. An edge stores pointers to the halo copies which reside on other processors. This array is used as the input to the SADT protocol, with the user functions processing each edge list in turn.

3.1 The counter-related SADT functions

Refering back to the SADT protocol stages in Section 2, Figure 3 describes the main user-supplied function required for the `Comms_CounterCompute` procedure. This relates to the computation of the send counter values, which describe the amount of actual data which will follow. For reasons of clarity, the other functions have been omitted here.

`User_CounterIndexing` will take the given edge list, and traverse the list of halos associated with each edge. The identifier of the processor holding each halo item, given by $halo \rightarrow proc$, will result in the increment of the associated send counter. This supports the computation of the counters within

Update the counters

```
void User_CounterIndexing (
PTETRAD_Edge *edge,          /* a list of mesh edges */
int *send, *recv)            /* the SADT data counters */
{
   PTETRAD_EdLnk *halo;      /* edge halo pointer */
   while (edge) {            /* inspect each edge */
      halo = edge → halo;    /* inspect the halos of the edge */
      while (halo) {         /* for each edge halo */
         send[halo → proc]++; /* the halo resides on processor halo → proc */
         halo = halo → next; /* go on to the next halo */
      }
      edge = edge → next;    /* go on to the next edge */
   }
}
```

Fig. 3. SADT counter-related user function

`Comms_CounterCompute`. The `Comms_AllToAll` call then initiates the communication of the counters. At this stage, each processor now has access to the amount of data which it will be receiving from the other processors, and the amount that it will send to them.

3.2 The data transmission SADT functions

Figures 4 and 5 provides a description of the user functions required for the `Comms_Multicast` procedure in the SADT consistency protocol. This relates to the packing, communication, unpacking and processing of the actual mesh edges. Once again, only the key user functions are given, for reasons of clarity.

Figure 4 shows the initial packing stage. `User_DatumSize` returns the size of the "flattened" data structure, which forms the contiguous memory area to be communicated. In this case, it is an address of a halo edge on a remote processor, and the new owner identifier to be assigned to it. These details are held in the variable *Comm*. For a particular processor, `User_PackDatum` is used to search a list of edges at a given mesh refinement level, and determine if any edge halos are located on that processor. Those halos are packed into a contiguous buffer area, ready for communication. The "Pack" function is a standard call within the SADT library, which will copy the data into the communication buffer. At this stage, the data buffers have been filled, and `Comms_Multicast` will carry out the communication between the processors.

3.3 The data reception SADT functions

Figure 5 shows the unpacking and processing stage. once the data has been exchanged. `User_UnpackDatum` will transfer the next data block from the communications buffer into the *EdgeOwner* variable. `User_ProcessDatum` will use the *ed* field to access the halo edge, and set its owner identifier to the new value.

Data type for communication

```
struct Comm_type {
   PTETRAD_Edge *ed;
   int own;
} Comm;
```

The size of the data type

```
int User_DatumSize ()
{
   return (sizeof(struct Comm_type));
}
```

Pack the datum into a buffer

```
void User_PackDatum (
PTETRAD_Edge *edge,     /* a list of mesh edges */
char *buf, int *pos,    /* storage space is available at buf[*pos] */
int pe)                 /* inspect all halo edges located on processor pe */
{
   PTETRAD_EdLnk *halo;                  /* edge halo pointer */
   int size = User_DatumSize();          /* the amount of storage required */
   while (edge) {                        /* inspect each edge */
      halo = edge → halo;                /* inspect the halos of the edge */
      while (halo) {                     /* for each edge halo */
         if (Edge_HaloHome (halo, pe)) { /* is the halo on processor pe ? */
            /* PACK THE HALO */
            Comm.ed = halo → edge;       /* note the halo's local address... */
            Comm.own = edge → owner;     /* on processor pe, and the owner */
            Pack (&Comm, size, buf, pos);/* pack this into the buffer area */
         }
         halo = halo → next;             /* go on to the next halo */
      }
      edge = edge →next;                 /* go on to the next edge */
   }
}
```

Fig. 4. SADT data communication functions: sending side

3.4 Comments

The use of SADTs to support the consistency of distributed mesh data has a number of advantages. The programmer is no longer concerned with how the communication of the data buffers takes place; a set of serial functions need only be defined in order to specify the particular mesh consistency procedure. In restructuring the redistribution phase of PTETRAD, the amount of code has also been substantially reduced (see Section 4.1). Additional reductions in code are available when SADT templates are used (see Section 5), which enable further simplification of the user functions. Since all sharing is carried out through the SADTs, the communications harness can be readily changed, without altering the application code (see the next section).

Unpack the datum from a buffer

```
void User_UnpackDatum (
void *in, char *buf, int *pos)
{
   Unpack (&Comm,
      User_DatumSize(), buf, pos);
}
```

Update the local mesh partition

```
void User_ProcessDatum (
void *in, void *out)
{
   (Comm.ed) → owner =
      Comm.own;
}
```

Fig. 5. SADT data communication functions: receiving side

Original PTETRAD version:

	Repartition + assign owners	Redistribute + establish links	Total
Appl.	1,780 / 53	7,300 / 216	9,080/269

SADT PTETRAD version:

	Repartition + assign owners	Redistribute + establish links	Total
Appl.	230 / 8	300 / 9	530/17
SADTs	440 / 11	1,660 / 40	2,100/51
Mesh			2,590/74
		TOTAL:	5,220/142

SADT libraries:

Update SADT	200 / 6
Exchange SADT	25 / 1
MPI Library	220 / 6
SHMEM Library	170 / 5
TOTAL	615 / 18

Table 1. A summary of the source code requirements (lines / KBytes)

4 Implementation Details and Performance Results

4.1 A summary of the source code requirements

Table 1 summarises the amount of source code in the original and new PTETRAD versions, for the mesh redistribution phase. The amount of code which the programmer must write has been reduced from 9,080 to 5,220 lines. A drastic reduction in the amount of application code has been achieved by supporting the stages of global mesh consistency as SADT calls, and implementing the mesh access operations within a separate library. The mesh access library is also being re-used during the restructuring of the solver and adaption phases. As a typical example, the code for the communication of mesh nodes during redistribution is reduced from 340 lines to 100 lines, with only around 20 of these lines performing actual computation.

Within the SADT library, the main *Update* SADT, for maintaining mesh consistency, contains 200 lines of code, and an *Exchange* SADT (for performing gather/scatter operations) contains 25 lines. The MPI and SHMEM communications harnesses each have their own library which support the `Comms_Function` operations (see Section 2 and below). New libraries can easily be written to exploit new high performance communications mechanisms, without any changes to the application code.

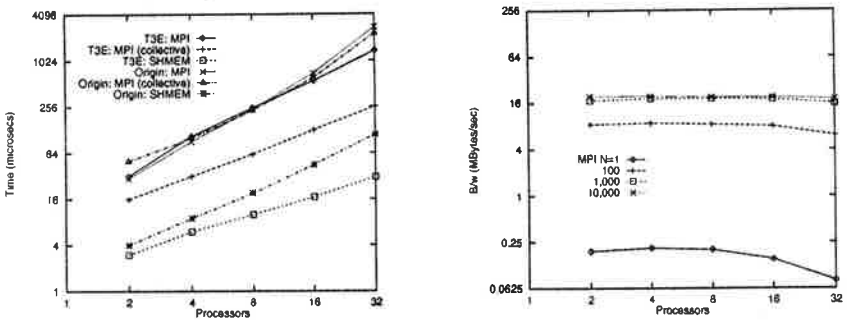


Fig. 6. (a) Comms_AllToAll; (b) Comms_Multicast: MPI on Origin 2000

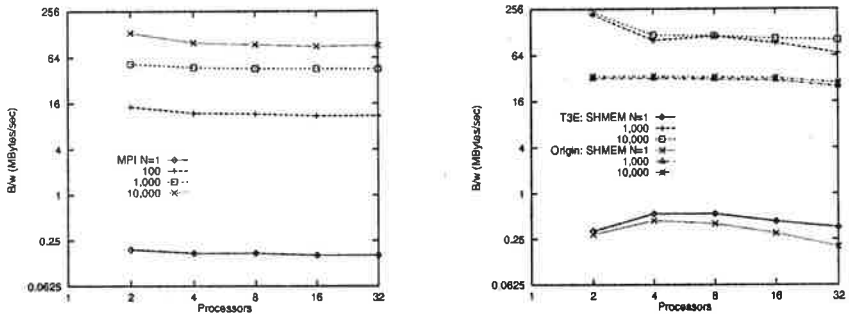


Fig. 7. Comms_Multicast: (a) MPI on Cray T3E; (b) SHMEM on both machines

Origin 2000:

	Adaption	Imbalance	Repartition	**Redistribute**	Imbalance	Solve
PTETRAD (4)	7.00	30 %	1.55	**3.29**	11 %	1.60
SADT-MPI (4)	6.97	30 %	1.52	**3.03**	11 %	1.60
SADT-COLL (4)	8.01	30 %	1.52	**3.04**	11 %	1.60
SADT-SHMEM (4)	6.96	30 %	1.53	**3.01**	11 %	1.60
PTETRAD (32)	3.01	26 %	0.49	**1.74**	13 %	0.20
SADT-MPI (32)	3.04	26 %	0.48	**1.72**	13 %	0.20
SADT-COLL (32)	3.02	26 %	0.47	**1.74**	13 %	0.20
SADT-SHMEM (32)	3.02	26 %	0.47	**1.73**	13 %	0.20

Cray T3E:

	Adaption	Imbalance	Repartition	**Redistribute**	Imbalance	Solve
PTETRAD (4)	5.87	19 %	0.99	**3.26**	11 %	1.23
SADT-MPI (4)	5.88	19 %	0.98	**3.04**	11 %	1.23
SADT-COLL (4)	5.89	19 %	0.98	**3.10**	11 %	1.23
SADT-SHMEM (4)	5.86	19 %	0.97	**2.78**	11 %	1.23
PTETRAD (32)	4.40	26 %	0.38	**2.12**	11 %	0.20
SADT-MPI (32)	4.38	26 %	0.38	**1.92**	11 %	0.20
SADT-COLL (32)	4.45	26 %	0.37	**1.92**	11 %	0.20
SADT-SHMEM (32)	4.44	26 %	0.37	**1.96**	11 %	0.20

Table 2. PTETRAD gas dynamics performance results (timings in seconds)

4.2 The SADT communications library

The communications operations employed by an SADT are supported by a small communications library, as outlined in Section 2 and above. This contains operations such as an all-to-all exchange (`Comms_AllToAll`), and the point-to-point exchange and processing of data buffers representing new or updated mesh data (`Comms_Multicast`).

Figure 6(a) shows the performance of `Comms_AllToAll` for the platforms being studied. On the Origin, the difference between using MPI send-receive pairs and the collective routine *MPI_Alltoall* is quite small. Pairwise communication performs better up to around 8 processors. Collective communications take 2,300 $\mu secs$ on 32 processors, as opposed to 2,770 $\mu secs$ for the pairwise implementation. On the $T3E$, pairwise communication performs very similarly, but the collective communications version performs quite substantially better in all cases (eg 258 $\mu secs$ down from 1377 $\mu secs$ on 32 processors), outperforming the Origin version. For both platforms, it can be seen that the pairwise implementation begins to increase greater than linearly, due to the p^2 traffic requirement, whereas the collective communications version stays approximately linear. A third implementation, using the SHMEM library, outperforms pairwise communication by at least an order of magnitude, due to its very low overheads at the sending and receiving sides, taking 30 $\mu secs$ on 32 processors for the T3E, and 110 $\mu secs$ on the Origin. In PTETRAD, this stage of communication represents a small fraction of the overall communications phase, so it is not envisaged that alternative implementation approaches will have any real impact on performance.

Figures 6(b) and 7 show the performance of `Comms_Multicast`, in which each processor i exchanges N words with its four neighbours $i-2$, $i-1$, $i+1$ and $i+2$ (in the case of $p \leq 4$, exchange occurs between the $p-1$ neighbours) (the user (un)packing and processing routines are null operations). On the Origin, performance reaches a ceiling of around 20 MBytes/sec for $N = 1000$ or larger using MPI, and 33 MBytes/sec using SHMEM, across the range of processors. For very small messages, the overheads of MPI begin to have an impact as the number of processors increase. On the $T3E$, the achievable performance using MPI was significantly higher, supporting 88 MBytes/sec on 32 processors, for large messages. Using SHMEM, this increases to 103 MBytes/sec, as well as improving the performance for smaller messages. Since this benchmark is measuring the time for all processors to both send and receive data blocks, the bandwidth results can be approximately doubled in order to derive the available bandwidth per processor. PTETRAD typically communicates messages of size $10K - 100K$ words, by using data blocking, and the above results show that this should make effective use of the available communications bandwidth.

4.3 PTETRAD performance results

A number of small test runs were performed using the original version of PTETRAD, and the SADT version of the redistribution phase, using pairwise MPI, collective MPI communication and SHMEM. A more comprehensive description

of the performance of PTETRAD can be found in [8, 9]. Table 2 shows some typical results on the Origin and T3E, for a gas dynamics problem described in [9], using 4 and 32 processors.

The results for the Origin show an 8% reduction in redistribution times on 4 processors, and 4% for 32 processors. The use of the SHMEM library doesn't improve performance any further in this case, since the high level of mesh imbalance mean that local computation is the dominant factor. Thus, the performance improvements when using the SADT approach originate from the tuning of the serial code. The other timings are approximately equal, pointing to the fact that the improved redistribution times are real, rather than due to any variation in machine loading. The T3E results show a reduction in times of between 7% and 10% using MPI, and a reduction of 15% on 4 processors by linking in the SHMEM communications library. The lower initial mesh imbalance, coupled with the very high bandwidths available using SHMEM, result in this significant performance increase. The slight increase in time on 32 processors using SHMEM seems to be due to a conflict between SHMEM and MPI on the T3E. When the complete PTETRAD code is restructured using SADTs, it is envisaged that this conflict will be removed.

The results show how performance can be improved using three complementary approaches. The use of an existing communications library, such as MPI, can be examined, to determine if alternative operations can be used, such as collective communications. Different communications libraries, such as SHMEM, can also be linked in. Finally, due to the clear distinction between the parallel communications and local computation, the serial code executing on each processor can also be more readily tuned. In the case of PTETRAD, the routines to determine the mesh data to redistribute were updated, to reduce the amount of searching of the local mesh partition. This shows up in the performance results by an immediate increase in performance when moving to the SADT version which still uses the MPI pairwise communications.

5 Conclusions and Future Work

This paper has described the use of shared abstract data types (SADTs) to structure parallel applications. This approach leads to a number of advantages:

- The resulting software is at a higher level of abstraction than, for example, message passing interfaces, since all explicit data sharing and synchronisation is encapsulated within an SADT.
- The clear distinction between the serial and parallel parts of the code allows greater scope for both the optimisation of the local computations and the performance tuning of an SADT on specific platforms.

In the case of the restructuring of the PTETRAD parallel CFD code described in this paper, some of the local data access methods were optimised, providing increased performance. In addition, the use of the Cray/SGI SHMEM communications library has allowed high performance SADT implementations on the Cray T3E platform. The amount of code has also been significantly reduced,

since the SADT used to support mesh consistency can be re-used in many parts of the code.

Currently, the mesh redistribution phase has been completed, with the solver and adaption stages due for completion in the near future. The redistribution phase has also made use of SADT *templates*, for further simplification. A template specifies many of the details of the `User_Function` operations, described in Section 2, given some assumptions about the input data set to be processed. In the case of the redistribution phase, the actual data to be redistributed is held as an array of processor identifier and local address pointer pairs, rather than having to traverse the mesh to determine the data. These increasingly higher levels of abstraction are aimed at eventually supporting the proposed SOPHIA applications interface [8, 9], which provides an abstract view of a mesh and its halo data, based around the bulk synchronous approach to parallelism [4].

References

1. C. Clemencon, B. Mukherjee and K. Schwan, *Distributed Shared Abstractions (DSA) on Multiprocessors*, IEEE Transactions on Software Engineering, vol 22(2), pp 132-152, February 1996.
2. D. M. Goodeve, S. A. Dobson, J. M. Nash, J. R. Davy, P. M. Dew, M. Kara and C. P. Wadsworth, *Toward a Model for Shared Data Abstraction with Performance*, Journal of Parallel and Distributed Computing, vol 49(1), pp 156-167, February 1998.
3. L. V. Kale and A. B. Sinha, *Information sharing mechanisms in parallel programs*, Proceedings of the 8th International Parallel Processing Symposium, pp 461-468, April 1994.
4. W. F. McColl, *An Architecture Independent Programming Model For Scalable Parallel Computing*, Portability and Performance for Parallel Processing, J. Ferrante and A. J. G. Hey eds, John Wiley and Sons, 1993.
5. J. M. Nash, P. M. Dew and M. E. Dyer, *A Scalable Concurrent Queue on a Message Passing Machine*, The Computer Journal 39(6), pp 483-495, 1996.
6. Jonathan Nash, *Scalable and predictable performance for irregular problems using the WPRAM computational model*, Information Processing Letters 66, pp 237-246, 1998.
7. P. M. Selwood, M. Berzins and P. M. Dew, *3D Parallel Mesh Adaptivity: Data-Structures and Algorithms*, Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, USA, March, 1997. CD-Rom, ISBN 0-89871-395-1, SIAM (1997).
8. P.M. Selwood, M. Berzins, J. Nash and P.M. Dew, *Portable Parallel Adaptation of Unstructured Tetrahedral Meshes*, Proceedings of Irregular'98: The 5th International Symposium on Solving Irregularly Structured Problems in Parallel (Ed. A.Ferreira et al.), Springer Lecture Notes in Computer Science, 1457, pp 56-67, 1998.
9. P.Selwood and M.Berzins, *Portable Parallel Adaptation of Unstructured Tetrahedral Meshes.* Submitted to Concurrency 1998.