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 or 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) e.g [1, 2] 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

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 651–658, 1999.

and optimisation can be made in the implementation of an SADT on a given platform. 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. 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 [3, 4]. 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 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 in which 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,

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

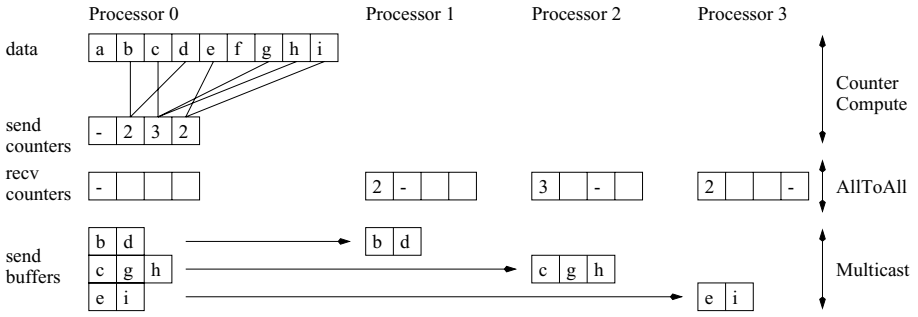

Fig. 1. The SADT communications stages

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)  /* interface with in and out data lists*/
{ int send[p], recv[p];    /* Counters used in communications.  */
  pre-processing;          /* Initialisation of internal data.  */
  communications preamble; /* Identifying data exchanges    ... */
  data communications;   /*  Exchanging data between 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, 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 1 shows an example of the operation of the protocol for a typical processor. For reasons of clarity the pre-processing and post-processing phases are 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

At the application level of PTETRAD [4], 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. The new mesh partitions at the base element level, are computed in the **repartition** stage using the parallel versions of Metis or Jostle. The local and halo owner fields for elements, edges, nodes and faces are updated in the **assign owners** stage. The data to be moved and the new halo data is communicated in the **redistribute** stage. Finally the **establish links** stage destroys 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

Pack the datum into a buffer

```
void User_PackDatum (PTETRAD_Edge *edge, char *buf, int *pos, int pe)
/* mesh edge list, storage space at buf[*pos], 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. 2. SADT data communication functions: sending side

as the input to the SADT protocol, with the user functions processing each edge list in turn.

The data transmission SADT functions `Comms_Multicast` makes use of the user function shown in Figure 2 . This is part of the SADT consistency protocol relating to the packing, communication, unpacking and processing of the actual mesh edges. Figure 2 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. Once the data has been exchanged the SADTs `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.

4 Implementation Details and Performance Results

The amount of source code in the original and new PTETRAD versions, for the mesh redistribution phase, was reduced from $9,080$ to $5,220$ lines, by using the SADT approach. A significant reduction in the amount of application code has also 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 adaptation 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 interfaces 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.

4.1 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 3(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.

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

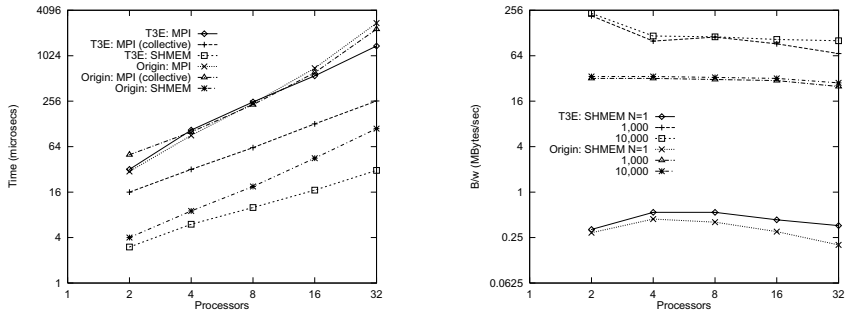

Fig. 3. (a) Comms_AllToAll; (b) Comms_Multicast: SHMEM

	Adaption	Imbalance	Repartition	**Redistrib-ute**	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 1. PTETRAD performance results (SGI/CRAY T3E:times in seconds)

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. Figure 3(b) shows the SHMEM results for both machines. 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.

4.2 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 [3, 4]. Table 1 shows some typical results on the T3E, for a gas dynamics problem described in [4], using 4 and 32 processors.

Results for the Origin (not given here) show an 8% reduction in redistribution times on 4 processors. The use of the SHMEM library doesn't improve performance any further in this case, since the high level of mesh imbalance means 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 in Table 1 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.

5 Conclusions and Future Work

This paper shows how performance can be improved using three complementary approaches. involving the use of shared abstract data types (SADTs) to structure parallel applications. The use of SADTs has made it easier to examine an existing communications library, MPI, to determine if alternative operations can be used, such as collective communications. Different communications libraries, such as SHMEM, have been linked in to provide high performance SADT implementations on the Cray T3E and SGI Origin 2000 platforms. 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. 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. 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. and some of the local data access methods were optimised. 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. Currently, the mesh redistribution phase has been completed, with the solver and adaptation stages due for completion in the near future. The intention is to support the proposed SOPHIA applications interface [3], which provides an abstract view of a mesh and its halo data.

References

[1] J. M. Nash, P. M. Dew and M. E. Dyer, *A Scalable Concurrent Queue on a Message Passing Machine*, The Computer Journal 39(6), 483-495, 1996.

[2] Jonathan Nash, *Scalable and predictable performance for irregular problems using the WPRAM computational model*, Information Proc. Letters 66, 237-246, 1998.

[3] 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 Comp. Sci., 1457, 56-67, 1998.

[4] P.Selwood and M.Berzins, *Portable Parallel Adaptation of Unstructured Tetrahedral Meshes*. Submitted to Concurrency 1998.