Using SADTs to Support Irregular Computational Problems

Jonathan Nash, Peter Dew and Martin Berzins
School of Computer Studies, The University of Leeds
Leeds LS2 9JT, West Yorkshire, United Kingdom

Abstract

There are well defined methods for supporting regular problems with scalable performance, typified by the HPF language and the BSP model. Less well understood is the solution of more irregular problems, supporting complex shared data structures and task dependencies, and typically requiring dynamic load balancing to sustain high performance. It is demonstrated how the use of Shared Abstract Data Types (SADTs), together with an extended BSP cost model, can support irregular problems in a structured manner. An SADT is an extension of a serial ADT which allows the concurrent invocation of its methods. A number of SADTs are used to implement a solution of the travelling salesman problem on the Cray T3D machine, and a description of the restructuring of a parallel CFD code using SADTs is provided, with initial results given for the Cray T3E.

1. Introduction

The Bulk Synchronous Parallelism (BSP) model [7, 13] provides a simple and elegant cost model, as a result of using supersteps to develop parallel software. The independent execution of processors in generating remote accesses allows a superstep to be costed by an $h-relation$ [7]. Specifically, given a machine with network performance g, barrier cost L and computational performance s, a superstep can be costed as $gh + sw + L$, where h and w are the maximum usage of each resource by a processor.

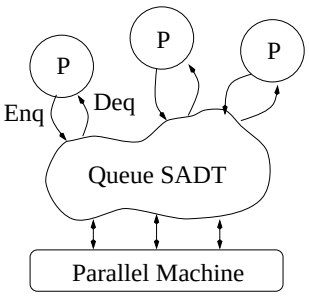
The BSP model seems less suited to the efficient support of irregular problems, which require dynamic load balancing and introduce runtime task dependencies [10]. This paper describes work on supporting irregular problems with scalable high performance, while preserving the BSP-style cost model. The key idea has been to develop scalable Shared Abstract Data Types (SADTs) which support dynamic sharing patterns [10, 3]. An SADT is an extension of a serial ADT which allows the concurrent invocation of its methods. It is shown how SADTs can encapsulate the often complex sharing patterns present in irregular computational problems, leading to portable code at a high level of abstraction, whose serial and parallel components can be more readily tuned for a given platform.

The next section provides an overview of SADTs. Two case studies are then used to demonstrate how SADTs can structure both parallel code and the costing analysis. Section 3 focuses on divide and conquer problems, using the travelling salesman problem, and develops a cost model to predict performance at design time, with results given for the Cray T3D. Section 4 describes the restructuring of a parallel CFD code, resulting in a very significant reduction in code complexity, while also providing increased performance, with results given for the Cray T3E machine. Section 5 points to some current and future work.

2. Shared Abstract Data Types

Shared Abstract Data Types (SADTs) [4] are used to describe the patterns of sharing among the processors, providing the algorithmic structuring advantages of ADTs, but with the ability to support concurrent access. SADTs can be used to encapsulate dynamic synchronisation and communication details, to provide both scalable and portable high performance [9]. The introduction of concurrency offers the possibility of weakening the sequential semantics, by allowing processes to observe different versions of the instance, where this will not effect the correctness of the application using it. Alternative SADT versions may be available, offering tradeoffs between consistency and performance, enabling a programmer to select the most efficient implementation which meets the correctness criteria [4].

There are many related research efforts in this area. *Distributed Shared Abstractions (DSAs)* [1] demonstrate the scalable and high performance on shared memory multiprocessors, using weak data consistency. *Parallel Abstract Data Types* [2] describe commonly used computational patterns in science and engineering applications. *Information Sharing Abstractions* [5] implement different ADTs which support concurrent access, with an emphasis on their ability to support compile-time and run-time optimisations, and in the development of performance and debugging tools.




```
Queue.Enq (data1);
Barrier ();

Queue.Enq (data2);
Barrier ();

data = Queue.Deq ();
```

Figure 1. A FIFO Queue SADT

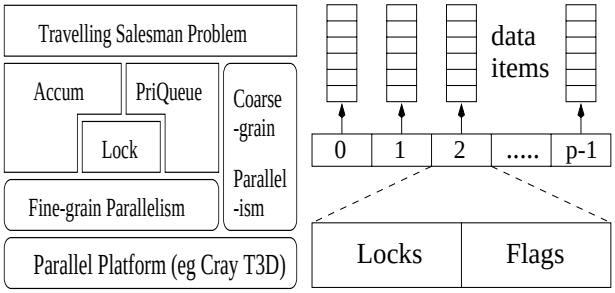


Figure 2. (a) Support for the TSP; (b) PriQueue implementation

2.1. The use of SADTs within a bulk synchronous environment

Coarse-grain supersteps organise parallel execution at the application level. Within a superstep, the processors access SADTs to implement the key sharing patterns of the application, which may use extended operations supporting fine-grain parallelism, to provide scalable high performance. Figure 1 shows a program using a FIFO Queue SADT [8]. Information is shared between the processors by enqueueing and dequeueing data items. In the code fragment (replicated on each processor), the dequeue into the variable $data$ will first access the enqueued items $data1$ from the first superstep, before any items $data2$ from the second superstep are accessed. Removing the barrier operations would result in an arbitrary interleaving of enqueue and dequeue operations among the processors (although the FIFO ordering still applies to the accesses generated by an individual processor).

2.2. The travelling salesman problem

The solution of the travelling salesman problem (TSP) presented here is a parallel version of Little's sequential branch-and-bound code [6]. The algorithm is based on related work carried out in the TallShiP project [4], which studied the application of SADTs to promote high level sharing mechanisms in parallel systems. The approach leads to the expansion of a tree of possible tours using a depth-first search, with the root of the tree representing all possible tours. The pseudocode in Figure 3 shows the parallel solution of the method. A Priority Queue (PriQueue) SADT holds the current set of generated tours. The length of the tour gives the corresponding priority of the item. An Accumulator (Accum) SADT notes the best tour discovered so far. Further details of the solution are given in [4].

```
PriQueue_SADT Tours;
Accum_SADT Best;

while (true) {
   task = Tours.PriDeq ();
   if (task_empty(task)) exit;
   if (task_len(task) < Best.Read ()) {
      if (task_leaf(task)) Best.Write (task);
      else {
         task_expand (task, &left, &right);
         Tours.PriEnq (left); Tours.PriEnq (right);
      }
   }
}
```

Figure 3. A solution of the TSP

Figure 2(a) presents an overview of the implementation method for the TSP. The approach is based around the support of both bulk synchronous parallelism and extended operations for supporting more irregular forms of parallelism, as mentioned earlier. Another common characteristic is that the code which solves the TSP is shielded from the increased complexity of the extended operations through the access of a set of SADTs, together with the use of coarse grain parallelism. The code given in Figure 3 shows that this particular solution in fact consists of a single superstep.

The implementation makes use of three SADTS - the Lock, Accumulator and PriQueue. The Lock is not directly visible to the application in this case, since it is used to support Accum and PriQueue. The implementation issues for the Lock and Accumulator are described in [3]. The PriQueue implementation is described in more detail in Section 3. One point to note is that the PriQueue stores the tasks within a highly concurrent shared data structure, and therefore requires a high performance network for good performance. An alternative implementation of the PriQueue, which is more amenable to networks where communication is more costly, is outlined in [3].

Operation	Cost
Swap/Inc	$D + 2g$
Get	$D + 4gx$
Put	$2gx$
Quiet	D
Wait	–
Barrier	L

Cost	Value
g	0.04-0.06
D	0.75-1.27
L	2.00
s	0.0533
t	0.0664

Figure 4. (a) SHMEM costing; (b) Cost values

3. A Performance Analysis of the Travelling Salesman Problem

A key characteristic for scalable performance was to use weakened data consistency [1]. Sequential consistency can provide a global priority ordering on the PriQueue elements, but typically through the use of a serialising lock. This becomes the bottleneck in large systems, limiting performance. The implementation here removes this lock by removing the strict ordering guarantee. Each processor $0..p-1$ holds a locally ordered data segment, as shown in Figure 2(b). These are accessed cyclically by the processors when storing and retrieving data, distributing the priorities approximately evenly. A processor is then only guaranteed to remove one of the p highest priorities (Section 3.4 will show that this does not have any substantial effect on the operation of the algorithm). This approach provides for a highly concurrent implementation, allowing scalable performance characteristics as the number of processors grow.

3.1. Performance at the machine level

As described in the introduction, a superstep cost can be modelled as $gh + sw + L$, where g, s and L cost the machine operations (network access, local computation and barrier synchronisation respectively), with h and w specifying their maximum usage by any one processor within the superstep. In order to effectively model the operations for fine-grain parallelism, two additional machine costs, D and t, are required. The cost D measures the round-trip network latency, and is incurred when there exists a data dependency within a superstep (for example, when accessing the results of an atomic increment). The cost t represents the time to transfer words within the local memory, and is present due to the fact that irregular applications typically access complex local data structures. So the cost of a superstep can now be represented by $gh + sw + tn + Dc + L$, where n and c again reflect the maximum resource usage.

3.2. Performance of the SHMEM library

For the implementation study on the Cray T3D, the SHMEM communications library of operations has been used, as given in Figure 4(a). SHMEM allows the direct access of remote memory locations through the specification of a processor-address pair, supporting high bandwidth and low latency operations. Coarse-grain parallelism can be supported through the use of the *Put* and *Get* operations, which write to and read from remote memories respectively, and a *Barrier* operation. *Put* supports the pipelining of multiple requests, to amortise the network latency cost (and allow overlapping of communication with subsequent local computation), with *Quiet* being used to suspend until these complete.

Figure 4(a) shows how the SHMEM operations are costed. For the Swap and Inc operations, performance is measured by the latency term D and the network access cost g. For the Put and Get operations, the time g is incurred at both the sending and receiving ends. Figure 4(b) gives each of the cost parameters for the Cray T3D. Further details on the costing approach can be found in [10].

The PriQueue concurrency can be exploited by the SHMEM operations which support fine-grain parallelism. Looking back to Figure 2(b), the local data segments are accessed cyclically by the processors, which can be supported by a SHMEM concurrent atomic increment operation, *Inc* (taken modulus p to obtain the required PriQueue data segment). Each data segment contains a number of locks and local flags, which are used to control the access to the data segments [3]. The implementation of these locks, through a lower level Lock SADT [3], can use the SHMEM atomic *Swap* operation, in which a new value is exchanged with the current contents of a shared word on a given processor. This operation allows for the construction of a scalable linked list [8]. In addition, *Wait* can be used to support fine-grain point-to-point synchronisation methods, which are typically used to coordinate the access of the shared lists [8].

3.3. Performance at the SADT / TSP levels

As described in Section 3.1, the performance at the machine level can be modelled using the terms $gh + sw + tn + Dc + L$, where h, w, n and c represent the maximum usage of each associated machine resource. When characterising the performance of parallel software, two terms are used to model the workload at a given level of abstraction. The term tot measures the total usage of a given resource across all processors, whereas max measures the maximum resource usage by any given processor. For example, the performance of the solution to the travelling salesman problem can be described as:

PriEnq (max,tot) + PriDeq (max,tot) +
Read (max,tot) + Write (u) + Compute (max)

In this case, tot and max refer to the number of generated tours. The costs $PriEnq$ and $PriDeq$ are for the PriQueue, and $Read$ and $Write$ the Accumulator (the u term has been

Operation	Cost
PriEnq (max,tot)	Inc(max,tot) + 882*max*s + Get(1,4*max,4*max) + Put(10,max,max) + max*Add_Heap(HEAP_SIZE) + 2 * Acquire(max,tot/p) + 2 * Release(max,tot/p)
PriDeq (max,tot)	Inc(max,tot) + 1107*max*s + Get(1,max,max) Get(9,3*max,3*max) + Put(1,max,max) + max*Sub_Heap(HEAP_SIZE) + 2 * Acquire(max,tot/p) + 2 * Release(max,tot/p)

Operation	Cost
Add_Heap (n)	Put(18,n,n) + 2*Get(9,n,n)
Sub_Heap (n)	Put(18,n,n) + 3*Get(9,n,n)

Operation	Cost
Acquire (max,tot)	Swap(max,tot) + Swap(max,max) + 37*max*s
Release (max,tot)	Put(1,tot,tot) + 36*tot*s

Operation	Cost
Swap/Inc (max,tot)	max*D + tot*2*g
Get (x,max,tot)	max*D + tot*4*g*x
Put (x,max,tot)	max*D + tot*2*g*x

Table 1. The PriQueue and related costing

found by experiment to be a small constant number). A $Compute$ term measures the local work required for a given number of tours. In this example, SADT access occurs within a single superstep. Processors independently produce and consume tours through the access of the PriQueue. The only exception is when all items have been temporarily removed from the PriQueue and new items are being enqueued (causing any idle processors to momentarily suspend execution), and at the termination stage. The former case typically only occurs for a short time at the start of execution, when the first few PriQueue items are being generated. Thus, the independent superstep behaviour is preserved at the application level. The highly concurrent behaviour of the SADTs preserves this state of affairs, allowing the simple BSP cost model to be employed.

The top set of costings in Table 1 gives the costing of the $PriEnq$ and $PriDeq$ operations. These include access to the *Inc*, *Put* and *Get* SHMEM operations. The *Inc* cost represents a potential source of contention, including a term of tot, reflecting its use to coordinate the access of the local data segments which make up the PriQueue. The PriQueue costing also includes the need to maintain a local priority ordering within each local data segment after an $PriEnq$ or $PriDeq$ occurs, using Add_Heap and Sub_Heap respectively. These costs are parameterised by the size of the local

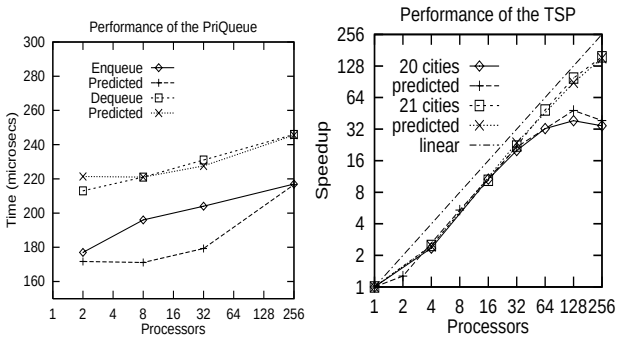


Figure 5. SADT and TSP performance results

segment (2^{HEAP_SIZE} items). Also included are costs to $Acquire$ and $Release$ a lock at a local segment. The term of tot/p in these costs reflects the scalable distributed implementation, in which the local data segments are accessed cyclically by the processors.

The next two costings in Table 1 give the costs for maintaining the priority ordering of the data segments and the Lock SADT. The ordering uses a binary search to exchange data items within the segment, resulting in the $HEAP\_SIZE$ term. The Lock SADT implementation is too involved to describe here, but further information can be found in [10]. Finally, the bottom table gives the cost of the SHMEM operations, when parameterised by max and tot. It can be seen that when a *Swap* operation is concurrently performed on a given shared word, the max term gives the usage of the network latency resource D (since $max * D$ gives the total time that any processor incurs the latency resource), and the tot term gives the usage of the network resource g (since $tot * g$ is the total contention at the memory module holding the word being accessed).

3.4. Predicted and observed performance

Figure 5(a) shows the close correspondence between the predicted and observed PriQueue performance, in which all processors continuously generate $PriEnq$ and $PriDeq$ requests. Figure 5(b) shows the TSP performance for 20 and 21 city problems. Speedups of over 150 on 256 processors for a 21 city problem are typical, reducing the time from 53 secs on one processor to $\frac{1}{3}$ sec on 256 processors. The predicted times are again close, using the performance models derived for the SADTs. The values for tot and max, described in Section 3.3, were taken from actual runs of the TSP solution, on a given number of processors. It has been demonstrated in [3] that the PriQueue supports good dynamic load balancing properties, and that the SADT overheads increase slowly with the number of processors, due to the scalable implementation approach.

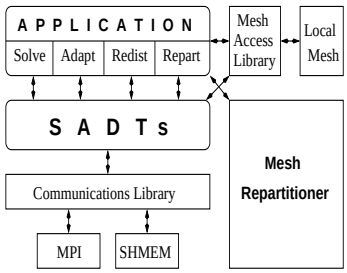


Figure 6. The new CFD code structure

4. The Application of SADTs to a CFD Code

This section describes work [1] investigating the use of SADTs to support problems in computational fluid dynamics [11, 12]. An unstructured 3D tetrahedral mesh, which forms the basis for a finite volume analysis, is partitioned among the processors, with frequently used remote mesh objects 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.

4.1. The restructuring of the code

The SADT to maintain the consistency of the mesh partitions uses two basic phases of communication. An all-to-all communication describes the amount of data which will be entering and leaving each processor, followed by the communication of the actual data between the mesh partitions. Both phases can be parameterised by a set of serial user-defined functions, so that the SADT can decide what specific mesh data fields to pack into a communications buffer, how to unpack any data received, and any subsequent processing (ie local mesh updates) which is required.

The new code structure is shown in Figure 6. At the application level, local mesh access is supported by a library of mesh routines. The mesh redistribution strategy is supported by linking in a parallel mesh repartitioner (such as Metis or Jostle). The global mesh consistency is handled by making calls to the SADT, which also performs local mesh updates through the mesh library. The code now operates in a sequence of supersteps, in the same manner as for the solution of the TSP in the previous section. The coordination between processors is supported by a small communications library which supports common traffic patterns.

The new SADT-based approach uses MPI, so that it may be readily ported between different platforms, and also 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. Using an alternative communications mechanism is requires a new communications library (typically around 250 lines of code), and linking the compiled library into the main code.

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

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:

Mesh SADTs	250 / 6
MPI Library	220 / 6
SHMEM Library	170 / 5
TOTAL	615 / 18

Table 2. A summary of the source code requirements (lines / KBytes)

4.2. Source code characteristics and performance results

A CFD code called PTETRAD [11, 12] was used to demonstrate this approach, which currently uses MPI. Table 2 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 reused 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 SADTs for maintaining mesh consistency contain 250 lines of code. The MPI and SHMEM communications interfaces each have their own library.

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, on 4 processors of the Cray T3E. The results are for a gas dynamics problem described in [12]. Improved performance of between 7% and 10% using MPI, and 15% by linking in the SHMEM communications library were found. A more comprehensive description of the performance of PTETRAD can be found in [11, 12].

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, new routines to determine the mesh data to redistribute reduced 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 using MPI pairwise communications.

5. Concluding Remarks and Future Work

This paper has presented an approach for the design and performance analysis of scalable high performance parallel software, focusing on the area of irregular computational problems, using Shared Abstract Data Types (SADTs) to support the key data sharing patterns. It was shown how applications could be supported which operate using coarse-grain parallelism, while making use of SADTs to support dynamic patterns of sharing, and to encapsulate possibly highly concurrent implementations. A solution of the travelling salesman problem on the Cray T3D was used as the main case study. This was used to demonstrate how SADTs can encapsulate highly concurrent implementation methods to give scalable performance, and to show how an extended BSP cost model can be used to predict performance.

A second case study was based around the restructuring of the PTETRAD parallel CFD software [11, 12], using SADTs. The approach has led to significant reductions in code complexity, through code re-use, and improved performance through the ability to more readily optimise the serial and parallel code. These increasingly higher levels of abstraction are aimed at eventually supporting the proposed SOPHIA applications interface [11, 12], which provides an abstract view of a mesh and its halo data, based around the bulk synchronous approach to parallelism [7]. Alongside this work is an ongoing effort to develop a cost model for the SADTs and PTETRAD. Within PTETRAD, a cost model could predict when the overheads introduced by mesh redistribution would be outweighed by the resulting improvement in performance of the solver, for example.

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] J. Darlington and H. W. To, *Building Parallel Applications Without Programming*, Abstract Machine Models for Highly Parallel Computers, (eds J.R. Davy and P.M. Dew) Oxford University Press, pp 140-154, 1995.

[3] P. M. Dew and J. M. Nash, *The High Performance Solution of Irregular Problems*, MPPM'97: Massively Parallel Programming Models Workshop, Royal Society of Arts, London, November 1997 (to be published in IEEE Press).

[4] D. M. Goodeve and S. A. Dobson and J. M. Nash and J. R. Davy and P. M. Dew and 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.

[5] 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.

[6] J. D. C. Little, K. G. Murty, D. W. Sweeney and C. Karel, *An Algorithm for the Travelling Salesman Problem*, Operations Research 11, pp 972-989, 1963.

[7] 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.

[8] 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.

[9] J. M. Nash, P. M. Dew and J. R. Davy, *A Parallelisation Approach for Supporting Scalable and Portable Computing*, Euro-Par'97, Passau, Germany, pp 678-682, August 1997.

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

[11] 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.

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

[13] L. G. Valiant, *A Bridging Model for Parallel Computation*, Communications of the ACM 33, pp 103-111, 1990.