University of Leeds
**SCHOOL OF COMPUTER STUDIES**
**RESEARCH REPORT SERIES**
Report 97.15[1]

# Towards a Model for Shared Data Abstraction with Performance[2]

by

**D M Goodeve[3], S A Dobson[4], J M Nash, J R Davy,**
**P M Dew, M Kara & C P Wadsworth[5]**

April 1997

[1]The work reported in this paper was supported by the UK Engineering and Physical Sciences Research Council under the project 'TallShiP: High-level sharing for parallel programming'.
[2]This paper has been submitted for publication in Journal of Parallel and Distributed Computing
[3]Now at Department of Computer Science, University of York, York, North Yorkshire, Y01 5DD, UK
[4]Well-founded Systems Unit, CLRC Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire OX11 0QX, UK
[5]Formerly at CLRC Rutherford Appleton Laboratory and now an independent consultant

**Abstract**

This paper demonstrates the utility of typed shared data abstractions as an effective high-level means of structuring and coordinating parallel programs. Access to data shared by concurrent processes is expressed through operations of Shared Abstract Data Types (SADTs). SADTs abstract low-level concerns of communication and synchronisation. The exposition addresses two major challenges: mismatches between representations and actual patterns of usage; and over-specified coherence. A prototype library of SADTs provides a set of implementations for each SADT, supporting stereotypical usage patterns and allowing exploitation of weakened coherence protocols. The efficacy of this approach is demonstrated on both a network of workstations and a dedicated massively parallel computer. A fine-grained irregular parallel computation obtained a speedup of 156 on a 256-processor Cray T3D, showing that the high levels of abstraction afforded by SADTs are compatible with efficient and scalable implementation.

# 1 Introduction

Writing codes for parallel machines involves excessive effort compared with sequential programming, due to the complexity arising from the combination of application code and the *assembly-level* operations generally used to co-ordinate parallel execution. To simplify program development, abstraction is essential to enable clean separation of application code from the low-level mechanisms, and encapsulating the issues of communication, synchronisation and distribution.

It must be remembered that the major reason for applying parallelism is to obtain high performance. Abstraction must be balanced against efficiency; to be effective, techniques should introduce the minimum of runtime overheads. The goal is therefore to construct an abstraction system, allowing the programmer to operate at a high level, yet implementing the abstractions in the most efficient low-level form exactly matching the needs of the application.

For regular parallel algorithms, the problem of efficient abstraction has largely been solved by languages such as as High-Performance Fortran (HPF)[21]. Irregular applications however pose problems for this regular model, and are therefore a largely unsolved area[2].

This paper presents an effective approach to high-performance parallel abstraction, based on Shared Abstract Data-types. The approach delivers both a high level of abstraction to the application developer, and simultaneously allows high and scalable performance to be obtained by exploiting the available optimisation routes.

This study is founded on the realisation that the issues arising come together around the interface of the operations that are meaningful for a data type, as in the early development of the notion of abstract data-types in eg. Simula67[19], CLU[30] and Alphard[44]. Extensions to a full object-oriented approach, ie. with inheritance may indeed be possible, but are beyond the scope of the present study, as the interaction between optimisation and inheritance has yet to be fully understood.

Thus, the entities that are used in this study are succinctly described as Shared Abstract Data-types. This provides the basis for the development of a sound semantics. In this paper, the utility of the SADT approach is demonstrated in practice.

An important approach to effective abstraction for irregular algorithms is through the use of typed shared memory. Several authors have developed typed shared memory systems, for example: Bal, Tannenbaum and Kaashoek's Orca language[6, 43], Chien and Dally's Concurrent Aggregates language[15, 16], Parkes, Chandy and Banerjee's ProperCAD library system[14, 39], and the MULTIPOL project of Yelick et al[13, 42]. These studies illustrate both the complexity of shared abstract type design, and the efficiency problems arising from mismatching type implementations to applications. This paper builds on these studies and demonstrates that efficient shared abstract types can be constructed and matched to application requirements.

The system introduced in this paper is based on a library of general purpose, re-usable abstractions such as queues, bags and shared variables, collectively referred to as *Shared Abstract Data-types*, or SADTs. The SADT approach splits the problem of parallel application programming into an application-oriented effort, coding against the abstract SADT interfaces, and a system-oriented effort producing the required SADT implementations. By designing general-purpose SADTs, the intention is that the complex system-oriented effort may be amortised over re-use of SADTs between applications. Although the creation of SADTs is relatively complex, the final implementations may encapsulate a wide range of optimisations including design idioms, weak coherence, task-specific representations and latency hiding. This allows a single programmer abstraction to offer high performance under a range of conditions. A fine-grained irregular parallel application is used to demonstrate the SADT approach on both a network of workstations and a massively parallel Cray T3D machine, demonstrating that very high, scalable performance can be achieved using this approach.

The simple use of an SADT to support the sharing involved in a parallel application is demonstrated using an example task farm in section 2. Section 3 discusses existing typed shared memory systems and contrasts these with SADT approach. The design of prototype SADT systems for both a network of workstations and a massively parallel machine are then discussed in section 4. The behaviour of these prototypes is investigated in section 5, where they are used to support the same parallel Branch-and-Bound solver application for the Travelling Salesman Problem. Results are presented for both systems, demonstrating both the level of abstraction achieved, and the performance delivered. Finally in section 6, conclusions are drawn about the SADT approach and its future development.

# 2 A Simple Task Farm Application

A common paradigm for solving problems that can be decomposed into independent sub-problems is *task farming*. The problem is divided into smaller sub-tasks which are solved separately in parallel and then combined to form the overall solution. Dividing the problem into many more tasks than there are processors allows load to be dynamically balanced by a *farmer*, which supplies new tasks from a pool in response to requests from idle *worker* processes.

A naive implementation places the farmer on one processor. This creates a bottleneck, and so to achieve scalability, the role of the farmer should be distributed. The revised task farm replaces the farmer with a shared queue, a simple example of a Shared Abstract Data-type, containing the pool of tasks to be solved, and a distributed task division mechanism shared amongst all the co-operating worker processes. The queue therefore provides the only sharing mechanism used in the application.

An outline of the task farm worker code is shown in figure 1. A worker takes a task from the queue. If it is too large to be solved immediately, the task is split in two; half is put back into the queue and the test is repeated on the remaining half. Once the worker has a sufficiently small task it solves it and goes back to

```
process Worker(pool:QueueSADT) { // all workers see same 'pool'
  while (not finished) {
    task := pool.deq();              // Obtain task from Queue
    while (size(task) > threshold) { // Subdivide (farmer)
      (task, task2) := subdivide(task);
      pool.enq(task2);               // Put half back
    }
  solve(task);                       // 'task' small enough...
  }
}
```

FIGURE 1: Pseudo-code for a worker process in the Queue SADT based Task Farm application

remove another task from the pool. All workers therefore participate in subdividing the original task and solving the resultant subtasks.

The performance of this application will be limited by the performance of the queue if the latency of `enq` and `deq` operations is too high. This may be addressed by substituting *different implementations* of the shared queue `pool`. The application was tested by applying it to the imaging of the Mandelbrot set, using three different implementations of the shared queue on a simulated massively parallel machine[23, 35]:

- Sequential. The queue is implemented in the memory of one processor in the system. All workers communicate with this memory to perform queue operations. The queue is protected from concurrent accesses using a lock.

- Concurrent. The queue is still implemented in a single memory, but now concurrent accesses are admitted; queue operations are implemented using wait-free primitives[27].

- Concurrent and Distributed. The representation of the queue is distributed across the entire parallel machine, with each processor hosting part of the representation. Enqueue and dequeue operations are then *striped* across the processors[23, 34].

The speedup curves for the simulated Mandelbrot application solving the same problem on different sizes

of machine is shown in figure 2. The image computed consisted of 512x512 pixels computed to 1000 iterations depth. The threshold size was set to 64 pixels, yielding 4096 individual tasks. The performance of

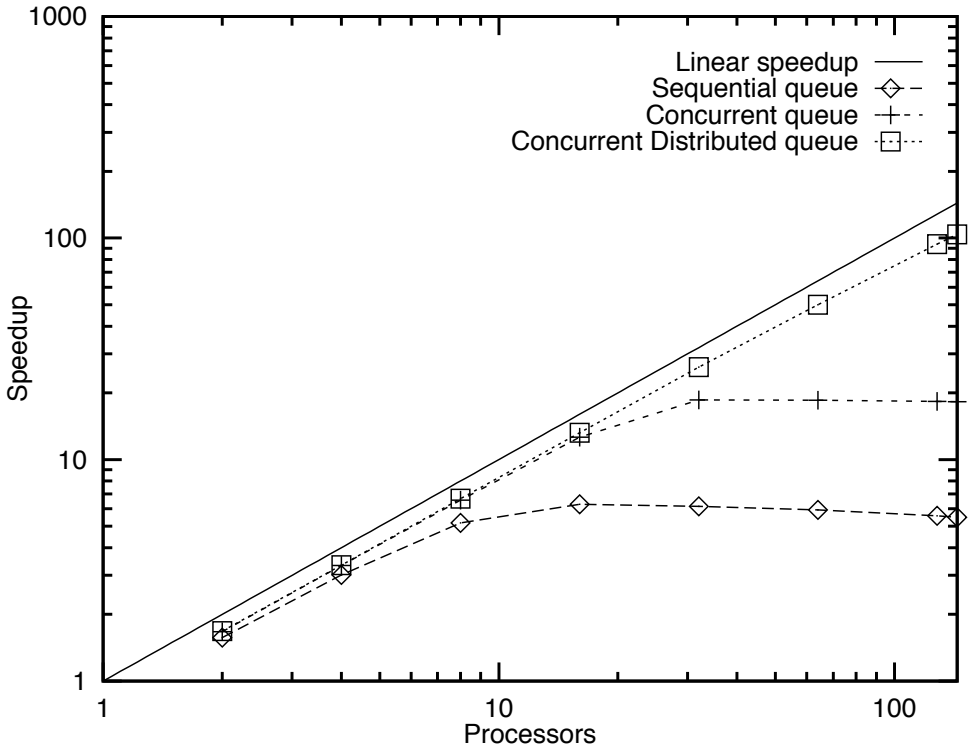


FIGURE 2: Speedup curves obtained for the Mandelbrot application running on a simulated massively parallel machine, using three different queue implementations.

the application with the sequential queue tails off below 10 as the number of processors is increased, due to contention of the single queue lock. A concurrent queue implementation can sustain a higher throughput, allowing speedups of the order of 20 to be achieved, but becomes limited by the access bandwidth to the processor memory holding the queue. As expected, distributing the queue has a marked effect, enabling the speedup of the application to scale beyond 100.

It is important to realise that the application code remains unchanged across the different queue implementations, demonstrating that the twin goals of abstraction and high performance are compatible. This has been achieved through re-implementing the shared queue below the abstraction boundary of its `enq` and `deq` operations. This hilights a route to achieving abstraction with performance; by optimising the imple-

mentation of shared data abstractions towards the requirements of applications whilst leaving the abstract interface unchanged. This theme is developed through the next section, introducing a system capable of exploiting this route.

# 3 Typed Shared Memory Systems

Types form the basis of much of modern software engineering in sequential computing[12]. Types hide the underlying implementation of data structures, providing an interface expressed in terms of the meaningful operations on the data rather than its low-level representation. Providing that the semantics of this interface remain unchanged, the underlying implementation can be changed without affecting the application. Types therefore decouple layers of a software system, simplifying interactions and allowing complexity to be controlled. Typed shared memory systems are designed to extend these benefits to a concurrent environment. This section discusses a variety of systems proposed in the literature, comparing and contrasting them with the SADT approach, outlined in section 3.4.

## 3.1 Synthesising Shared Types

Typed shared memories are populated by instances of types which can be accessed from across a parallel machine. There are two approaches to the production of these types; applying sharing mechanisms to essentially sequential types, and producing truly concurrent implementations.

### 3.1.1 Sequential Type Sharing

A number of systems provide mechanisms for producing shared types, or objects, from essentially sequential code. Typically, concurrency control is applied at the level of the operations on the sequential objects to maintain data coherence. A major source of inefficiency in these systems is caused by poor locality

when accesses to an object require network traffic. Another is contention for an often-used object caused by excessive locking.

The Orca language[6] employs two implementation styles to sequential objects; single instance and replication. A single instance object resides at one static place in the system, with all operations being forwarded to that location. A replicated object installs a copy of itself on every processor. Read operations occur against a thread's local replica; update operations are atomically broadcast to all copies. This is an important optimisation for objects which are updated only infrequently.

An approach to improving locality without excessive coherence traffic is to *migrate* objects, as adopted in the Emerald[10] and and DoPVM[26] systems. Whilst avoiding the false-sharing phenomenon of low-level Distributed Shared memory (DSM) systems[38], performance can still be compromised by excessive object movement. This is reduced in the DoPVM system by allowing read-only replicas of objects to exist, avoiding migration on read. DSM systems also employ alternative coherence styles to minimise the performance penalty involved in migrating data[7, 8].

Herlihy develops an approach to the synthesis of *wait-free* objects[28]. These deliver performance comparable with objects with sequenced accesses, however the main import of this work is the fault-tolerance arising from the wait-free approach.

### 3.1.2 Truly Concurrent Types

To achieve higher performance, effort must be focussed on the data structures and algorithms used to coordinate concurrent accesses. There is a significant body of literature on the design of high-performance concurrent data structures, for example [40, 41]. Generally these implementations rely for their efficiency on features of the underlying architecture. The Berkeley MULTIPOL project[13, 42] attempts to specifically address irregular applications by using specialised high-performance implementations of objects.

To provide a platform for implementation, systems have been proposed that implement a concurrent type as a community of actor-like[1] objects. Different threads access the same instance through different actor objects, which are then co-ordinated to present a single abstraction. There have been several examples of this approach in the literature[14, 15, 16, 17, 39], perhaps the most flexible being the Concurrent Aggregates system[15]. An aggregate consists of a co-ordinated community of *representatives*, each of which is a simple sequential object. An operation on an aggregate is forwarded to one of its representatives chosen at random, which performs the operation on behalf of the caller. This may give rise to an arbitrary amount of activity between the representatives of the aggregate. Concurrency comes from the interactions between the representatives rather than from within a single representative.

The effectiveness of Concurrent Aggregates has been demonstrated in [16], although the programming effort required is high and re-usability is largely not addressed. The ProperCAD library system[14, 39] is an attempt to employ similar techniques to build a re-usable library of types on workstation networks for the specific application domain of VLSI CAD.

The WPRAM model provides a platform for the implementation of high performance scalable types[35, 34]. The major benefit of this model is a well-founded costing methodology, allowing performance and scalability to be assessed accurately at the design stage.

## 3.2 Representation Mismatch

Poor performance often arises when the choice of the underlying implementation is mis-matched to its use. Taking an example from the sequential world, one common representation of lists is to create a linked collection of elements referenced from the head end. This structure is processed very efficiently by functions which iterate along the list from the head end; it is extremely inefficient when used in functions which process from the tail end. This is *not* a failure of type abstraction but simply an unwelcome interaction

between the hidden representation and a particular pattern of use of the type. When recognised in the sequential world, this mismatch leads programmers to choose more suitable implementations of types.

The situation is exacerbated by distributed processing. If the distributed representation of a shared type is poorly chosen for the use made of it, for example by placing a frequently-accessed piece of data at a remote location, the penalty can be significant network-induced delays.

High-Performance Fortran[21] allows the user to direct placement of elements of the only available shared type, the memory array, in order to match the location of data to its use. This optimisation relies on a small repertoire of placement strategies, such as regular and cyclic. This strategy is only suitable at present for regular problems, where uniform patterns of access can be identified in advance.

It may be observed that there is typically no single representation which is well matched to every pattern of use, and mis-match becomes especially severe when only a small repertoire of distributed implementation schemes are available[43]. This complicates the task of designing libraries of re-usable shared types, since each type may need multiple representations to avoid inefficiencies, and a way of choosing which to use under a given set of circumstances.

## 3.3 Coherence

Coherence concerns the views that different threads have of a single data item. Providing a totally uniform view of data to all threads, essentially making the view consistent with a single global operation order, becomes expensive as network latency increases[4, 5]. If coherence can be relaxed, it follows that overheads in maintaining the shared data abstraction may be reduced. This route has been pursued in several systems[7, 20].

At the present state of the art, recognising that weak coherence can be applied is largely down to observation of the algorithms involved. Earlier studies eg. [13] have demonstrated that coherence requirements

can be extracted for particular algorithms, providing a route to the implementation of weak types, and the specification of coherence requirements.

In general algorithms cannot tolerate this sort of relaxation of coherence in arbitrary variables; they may require some variables to be kept coherent whilst others may be relaxed. Although it is possible to build strongly coherent types out of weaker versions in an application, this severely complicates the programming task. There is therefore a case for providing a *mixed* model, where the relaxation in coherence of some types can be applied, whilst other types must be strictly coherent[3]. There is a case therefore for providing both strong and weak versions of a concurrent type in a library. Careful design is needed to ensure that such weakened types provide the programmer with useful and understandable abstractions. Indeed coherence may be perceived as a property ultimately of variables (rather than of memories or of types) with stronger or weaker coherence being chosen to suit the different variables.

## 3.4 Shared Abstract Data-types

Shared Abstract Data Types (SADTs) represent a synthesis of the experiences of existing systems into a flexible shared type framework suitable for high-performance applications. The intention is to develop a small set of shared types suitable for expressing the common sharing modes encountered in parallel algorithms, and which may be flexibly optimised and widely re-used. The experimental set currently consists of queue, priority queue and bag shared container types, and the accumulator[25] shared variable type. Others which have been identified but not yet characterised and implemented fully include grids, lists and trees. By maintaining a small set of types, the complex low-level implementation effort is recouped over repeated re-use of these types in parallel applications.

Each type in the library has a set of implementations which can be selected at compile-time. The signatures of all implementations of the same SADT are the same, but each is designed to best support either or both

of:

- a stereotypical usage pattern, improving a type's ability to support a particular mode of sharing; and
- a different level of coherence, allowing reduction of maintenance traffic.

For example, an implementation `WeakReplicatedPriQueue`, offering a relaxed coherence model and a replicated implementation of the `PriQueue` SADT can be chosen and bound in to an application expressed in terms of the `PriQueue` SADT.

Application experience is guiding the population of this set as new patterns of usage and coherence models are encountered. Thus the suite of representations available for SADTs is designed to evolve.

As far as interface design and usage is concerned, the specification of types is straightforward. The issue of usage patterns is orthogonal to such a specification. Coherence however complicates the picture, and is generally specified by reference to some notional sequential model of execution through for example linearizability or sequential consistency[29]. Specifying weaker forms of consistency is an ongoing research issue[13].

For the programmer, the SADT model provides of a set of re-usable polymorphic shared types which may be freely instantiated. An application is written with reference to an abstract interface. The SADT implementation required is then selected and bound into the application. The exact implementation chosen for a particular SADT instance varies according to the degree of coherence required, the expected patterns of use *etcetera*. It is therefore possible to deploy optimisations in a targeted way, and to encapsulate these, possibly complex techniques behind the abstraction boundary. This allows SADTs to provide wide-ranging and highly optimised support for parallel applications without exposing complexity to applications based upon them. The rest of this paper describes some implementation techniques for this model, and an example of using it to create an efficient and portable application.

# 4 Prototype SADT implementations

Prototype implementations have been developed for a network of workstations and for a massively parallel machine. This section briefly describes the systems and how SADTs are implemented on each.

## 4.1 Implementing SADTs on a Network of Workstations

The main prototyping environment used in this work has been developed on a ethernet-connected network of workstations. This type of system is challenging for parallel programming due to the significant overheads involved in communicating between address spaces on different machines. This requires the highly effective use of type, usage and consistency information to drive efficient SADT implementations.

The SADTs written for this platform rely on an underlying run-time executive, which provides low-level services such as SADT creation, initialisation and binding. The run-time executive and the hosted SADTs and applications are written using the Modula-3 language[37]. The language provides a strongly-typed object-oriented environment, with automatic garbage collection and embedded support for threads. In addition the standard library accompanying the Digital SRC Modula-3 system provides low-level interfaces into the language run-time that have enabled efficient implementation of the required primitives.

Modula-3 has proven to be an excellent research test-bed. However, a common criticism of such novel languages is that their comparatively immature and unoptimised compilers make performance extrapolation to real systems difficult. In practice, the performance of Modula-3 code compares well with that of C. For a particular application which makes heavy use of object creation and disposal, the speed of the Modula-3 application was only 23% less than the same application coded in C[6]. A study of the implementation of Modula-3 has demonstrated how the performance of compiled code may be further improved[18].

[6] Experiment conducted on an SGI Indy workstation, running IRIX 5.2. C compiled using gcc version 2.7.2. Modula-3 compiled using the DEC SRC compiler release 3.5.

Interaction between address spaces is facilitated by the Network Objects system[9]. This provides a lightweight and efficient RPC facility that underlies all interactions between the run-time executives in each address space, and across the representations of the SADTs.

### 4.1.1 SADT architecture

An SADT is composed of a set of representatives, one per address space in the system. The application code in each address space communicates with its local representative using a simple front-end object. To the application code therefore, the SADT appears to be a simple object and may be invoked in the same way; for example `queue.enqueue(newItem)`.

Operations on SADTs are composed of one or more *event* messages communicated between the front end object and the local representative.

```
PROCEDURE enqueue(self:T; in:Item.T) = (* method of front-end object *)
  VAR msg := NEW(Theory.Insert, item := in)  (* Build 'event' message *)
  BEGIN
    EVAL self.rep.perform(msg)  (* pass to local representative *)
  END enqueue;   (* 'perform' always returns a message, discarded. *)
```

In response to these messages, the representative can in turn generate event messages that are passed to other representatives of the same SADT in other address spaces. This is similar to the implementation style of the concurrent aggregates system[16] and the ProperCAD model[14], except that the *local* representative is always invoked.

This architecture is supported by a small run-time executive that supports SADT creation, binding, garbage collection and other simple services. The creation of an SADT involves building a representative in each address space, setting up their routing tables to communicate with the rest of the SADT implementation, and invoking any SADT-specific initialisation code. Representatives may also contain multiple threads,

allowing them to perform autonomous activity in the background, such as periodic load balancing.

The representatives act as a substrate for co-ordinating activity across address spaces. The architecture is versatile, allowing a wide variety of possible SADT implementations. Figure 3 shows three basic implementation styles realised using this architecture. The simplest SADT implementation involves all

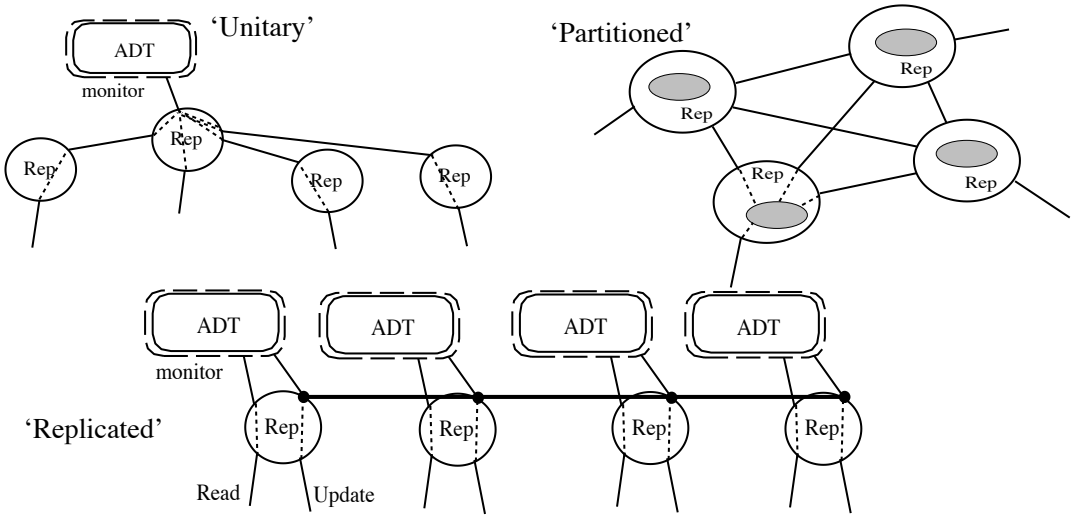


FIGURE 3: Three different implementations styles for SADTs using the representative architecture; shared object (centralised), replicated, and partitioned.

requests being forwarded to an object implementing the abstraction in a single address space. The code in the representative to handle this forwarding is of the form:

```
PROCEDURE perform(self:Rep; msg:Theory.T):Theory.T = (* centralised method *)
  BEGIN
    IF self.Root() # SADTRt.Host() THEN (* Not local - forward! *)
      RETURN self.GetRep(self.Root()).perform(msg)  (* RPC Call *)
    ELSE  (* Handle locally... *)
      ...
```

Whilst easy to realise, performance is bounded by both the network and the performance of the processor hosting the actual object. The second style is a replicated object, in which coherence is retained by a write-update broadcast protocol.

```
PROCEDURE perform(self:Rep; msg:Theory.T):Theory.T = (* replicated method *)
```

```
  BEGIN
    TYPECASE(msg) OF
      Theory.Update => BEGIN
        self.Broadcast(msg); (* Send to other reps and apply locally *)
        self.adt.Update(NARROW(msg,Theory.Update).item);
        RETURN msg (* Return to front-end object *)
      END
    | Theory.Rd => ...
```

Each of these styles delivers strong coherence: updates occur immediately and are immediately visible to all threads sharing the SADT. Weaker coherence, and thus reduced coherence maintenance traffic, may be achieved by buffering updates within the representative.

```
PROCEDURE perform(self:Rep; msg:Theory.T):Theory.T = (* distributed method *)
  BEGIN
    TYPECASE(msg) OF
      Theory.Update => BEGIN
        self.buffer.Append(msg); (* Buffer locally... *)
        RETURN msg (* Return to front-end object *)
      END
    | Theory.Sync => BEGIN
        self.buffer.Flush();  (* Apply updates en-masse *)
        RETURN msg
      END
    ...
```

Update operations therefore return directly to the caller after buffering the operation, avoiding network overhead. The buffered updates may then be applied by an explicit synchronisation, or periodically in the background.

The *partitioned* style is the most general. Here, the representation of the SADT is split between address

spaces. The representatives contain part of a complete data structure, organising the distribution of this structure transparently to the user. This third style gives the implementer most freedom to make use of individual type, usage and coherence information in design.

## 4.2 Implementing SADTs using Shared Memory primitives

This section considers the implementation of SADTs using the low-level shared memory primitives available on the Cray T3D platform. This leads to a quite different implementation style to that described above, but the nature of the tradeoffs that can be exploited in the design of types is very similar.

The prototype SADT system on the Cray T3D is coded in C, using the primitive operations of the WPRAM computational model[35], and the underlying Cray SHMEM Shared Memory primitives library[11]. These primitives are low-level but lightweight, and offer a well-founded cost model that can be used to drive the implementation of types. The nature of the network of workstations environment is such that an accurate cost model would be hard, if not impossible to produce.

The network of workstations SADT system makes use of Modula-3's extensive support for strongly-typed object-oriented and network programming. However, supercomputer applications tend to be based around highly optimising C and Fortran compilers. We have therefore chosen to implement the T3D SADT system using the common *object-oriented C* style of programming, building objects explicitly using C structures and associated functions, for example:

```
void QueueSADT_enqueue(QueueSADT self, void *data);
void *QueueSADT_dequeue(QueueSADT self);
```

This system will be re-engineered in C++ in the future. Currently, it shows that the SADT style can be adapted to a range of programming languages and styles.

### 4.2.1 WPRAM/SHMEM Implementation architecture

The T3D SHMEM library allows the memory on the machine to be addressed from any processor, using a processor identifier and a local address on that processor. Remote memories are accessed without interrupting the target processor. The basic operations within the SHMEM library allow the transfer of blocks of data between processor memories, atomic increment and atomic swap operations, and synchronisation on completion of outstanding operations and on the contents of particular locations. These primitives make it straightforward to implement the synchronisation structures required by the WPRAM, and for the implementation of SADTs. The only structure commonly used in SADT implementations is a lock. This has a simple implementation at present. Alternative locking algorithms offering better performance, such as the MCS lock[33] may be applied in future.

The prototype system statically binds the required SADTs into the SPMD program. This allows the SADT implementations to find the required co-ordination data structures at the same address on all processors. It is a simple future refinement to allow dynamic creation and binding using processor local tables which would also allow adoption of a more general program style.

The basic structure of an SADT on the Cray T3D is shown in figure 4. Local operations use static address knowledge to access data structures on the local and other processors in the system to perform SADT operations. The SHMEM substrate, with its synchronisation mechanisms is used instead of the set of representatives used in the network of workstations system. Remote operations are implemented by the calling processor however, accessing data across the network, rather than by a representative on the target processor. In the figure shown, an access to the local static data table indicates the processor at which the following operation should be performed. Remote dynamic data is accessed by indirecting through the remote static data table.

Atomic increment is implemented on the T3D using a high performance *annex register* mechanism. Each

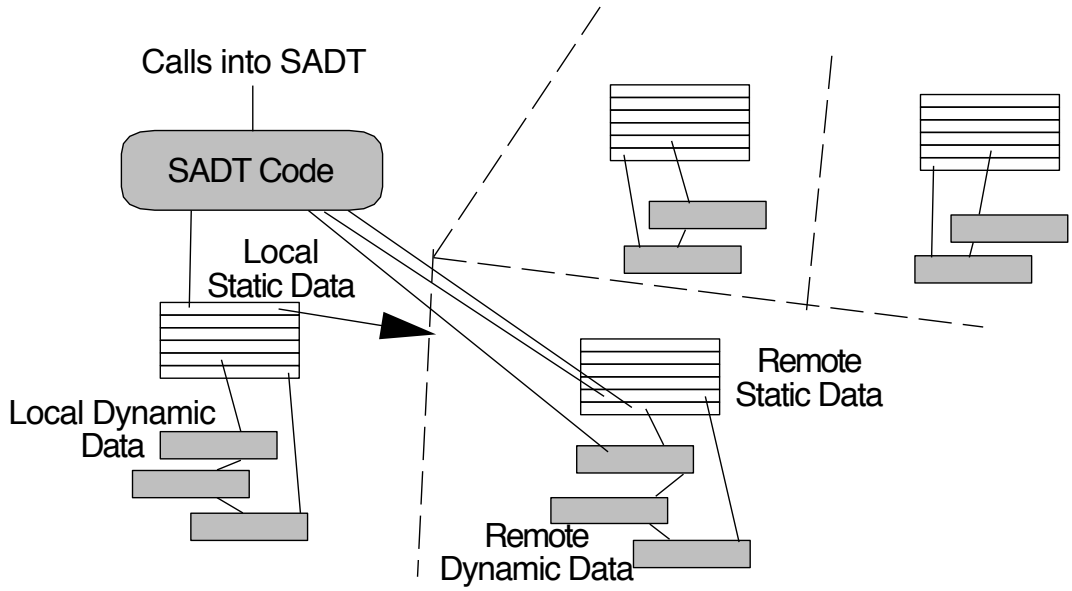


FIGURE 4: Illustration of and SADT implementation on the Cray T3D using the SHMEM substrate.

processor has one register that can be used to support atomic increment operations. To perform an atomic increment, a remote processor programs an annex register to point to the correct processor and register and then performs the operation referencing this. Any memory word can be used to support atomic swap operations.

A problem with remote access of dynamic data is that of garbage collection. Once a remote data item can be discarded, it needs to be freed on the processor hosting it. The current version of the system largely ignores this problem. To solve it, a runtime mechanism for maintaining reference counts could be used. This issue will be addressed in the object-oriented re-implementation of the system.

### 4.2.2 WPRAM cost-model

The WPRAM is an extension of the BSP model[32, 35], adding a small number of atomic operations such as swap and increment. Additionally, fine-grained synchronisation mechanisms based on tag variables and completion of outstanding operations are introduced which permit synchronisation, and thus data-

dependencies within super-steps to be expressed. These low-level operations can be characterised for a target architecture using a small number of architectural parameters, including network capacity, network latency, processor speed and machine size.

The WPRAM remote memory access operations map directly onto the SHMEM library. The additional primitives required are simply implemented using the SHMEM synchronisation mechanisms. The use of the WPRAM style for implementing SADTs allows the costing model to be applied to SADT implementations. Initial work on a queue SADT indicates that this approach yields both accurate and insightful results[36].

# 5 Example Branch-and-Bound Application

In this section, the application of SADTs to the development of a parallel application is demonstrated, showing the level of abstraction maintained, the optimisation routes exposed and the level of performance achieved. The Travelling Salesman Problem (TSP) is a well-known combinatorial optimisation problem. Given a set of cities and the distances between them, the problem is to find the shortest possible tour that visits each city exactly once. In the general case, the problem need not be symmetrical.

The parallel solver is developed from a sequential code implementing the Branch-and-Bound algorithm of Little et al.[31]. The algorithm proceeds by refining a description of the solution space as a tree of *nodes*. Each node represents a subset of the solution space, with the root of the tree representing the entire solution space. Each node represents a set of feasible tours. The tree is progressively expanded by splitting nodes into children containing smaller subsets of the problem space, and pruned by removing nodes that cannot be the ancestors of better solutions than are currently known. The algorithm terminates when the solution space has been exhausted.

The expansion of a node into its children is guided by a heuristic which makes it likely that the set of tours

represented by one child contains the optimum, and conversely that the set of tours of the other child does not. A second heuristic guides the algorithm to chose the most likely node to contain the optimal solution at the next execution step. A lower-bound on the length of any tour represented by a node is maintained by the algorithm. The heuristic guides the algorithm to next expand the node with the lowest lower-bound of all those available. This is achieved through the use of a priority queue to store the nodes of the tree, where the maximal priority corresponds to the lowest node lower-bound.

A leaf node of the tree represents a single tour. Once a tour is found, its length is used to *improve* an upper-bound on the length of the optimal solution. If the solution just found is shorter than the best currently known, then the best known solution is updated. Any node whose lower-bound exceeds the length of the currently best-known tour can be discarded, as it cannot possibly contain a better solution.

## 5.1 Parallel TSP code

The parallel application consists of a number of identical worker processes that run asynchronously, only co-operating through the use of SADTs. A worker repeatedly removes a node from the priority queue and then performs one of the following:

- Expansion. Generates two child nodes, putting them back into the queue.
- Solution update. If the node is a complete solution (leaf), then the best-known solution is updated.
- Pruning. If the node removed from the queue has an excessive lower-bound, it is discarded.

An outline of the worker code is shown in figure 5.

Two SADTs are used; a priority queue `store` storing the current set of nodes, and an *accumulator* SADT `shortest` storing the best known tour. The best-known tour is stored in the accumulator, which is an abstraction of a variable with an update function[25]. The accumulator can be read to return the current

```
solve(store:NodePriQueueSADT, shortest:TourAccumulatorSADT) {
  loop until exit {
    node = store.dequeue();
    if (node = NULL) exit;
    lowest = shortest.Read().length;     // Shortest so far
    if (node.length < lowest) AND (NOT node.leaf()) {
      (left, right) = node.expand();
      store.enqueue(left.lowerbound(), left);
      store.enqueue(right.lowerbound(), right);
    } else {
      if node.leaf()
        shortest.update(node.tour());  // Combines tour and length
      discard node;
    }
  }
}
```

FIGURE 5: Object-based Pseudo-code for a TSP solver worker process. The C code for the Cray T3D and the Modula-3 code for the network of workstations are both very similar to this pseudo-code fragment.

value, or updated in which case the update function (in this case a minimum function) is applied to the current accumulator value and the argument of the update operation. The new value of the accumulator is the return value of this function. By combining the tour with its length as the accumulator stored type, the best known solution is also retained on a successful update.

This application is similar to other parallel branch-and-bound solvers proposed in the literature[22]. The contribution of this solver is the abstraction from the low-level details of data management that it achieves, and the clean expression of the core algorithm.

## 5.2 SADT Optimisation

The application consists of a set of identical worker processes, each running on one processing node in the target system. The SADT-based optimisation for each of the target platforms is described below.

The application was first implemented on a network of workstations, using the runtime system and SADT library described in section 4.1. The first implementation used three SADTs: `shortest`, a blocking

priority queue SADT `store`, and an additional accumulator `counter` implementing a simple integer counter to facilitate termination detection. The simplest implementation style for these SADTs was the unitary style (figure 3). The high network traffic generated by this style of implementation resulted in very poor application performance. An execution profile of the application (figure 6) clearly demonstrates why.

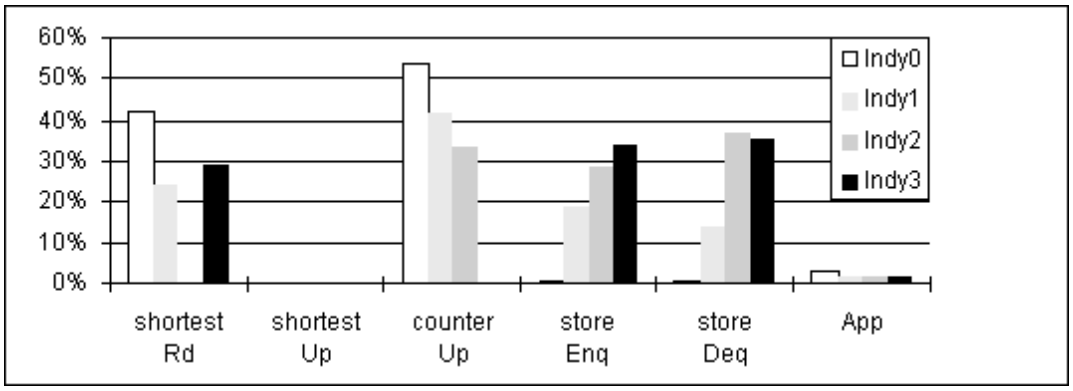


FIGURE 6: Execution time spent performing each SADT operation, and the remaining time for executing the application for an example 20-city problem solved on a network of 4 SGI-Indy workstations connected using TCP/IP over a 10MBit/sec Ethernet.

The profile shows the vast majority of the execution time is spent performing operations on the SADTs, leaving only a few percent for executing the remaining application code. It is not surprising therefore that this application *slows down* on multiple processors. The high operation cost is generated by the synchronous network traffic required by all SADT operations. These costs are clearly excessive for this granularity of computation, resulting in very poor performance.

A common route to improving performance in this situation is to modify the application to increase granularity, and thus reduce the dominant effect of network delays. Instead, the two identified optimisation routes were exploited: usage and coherence. Additional performance benefits were obtained by carefully considering the semantics of the interface to the priority queue SADT.

### 5.2.1 Replicated Accumulator

The accumulator `shortest` holds the length of the shortest known complete tour. Each time around the main loop of the algorithm, the accumulator is read. It is only updated however when a complete solution is found. This occurrence is relatively infrequent; typically updates occur less than ten times during a complete run. The execution profile in figure 6 clearly supports this observation; the proportion of execution time spent updating the accumulator is negligible.

To exploit this usage pattern, the implementation of the accumulator was re-written in the *replicated* style depicted in figure 3. Each local representative contains a cached copy of the current value of the accumulator SADT, which is used to service local *read* operations. Each copy is kept consistent with all *update* operations by broadcasting operations across all representatives This is an expensive, but relatively infrequent operation. The replicated implementation makes read operations about two orders of magnitude faster on a network of workstations than the unitary implementation, at the expense of making update operations about an order of magnitude slower.

The new implementation of the accumulator has the same signature as the unitary accumulator, and the same coherence properties. The replicated accumulator can therefore be directly substituted with no change to the application code.

### 5.2.2 Partitioned 'Shuffling' priority queue

The priority queue supports equal numbers of enqueue and dequeue operations over the execution time of the algorithm. The worker processes in all address spaces also behave similarly, hence there is no simple usage pattern that can be identified.

The sequential algorithm ensures that the best node known is examined at each step. If however the next best node is *not* examined next, the correctness of the algorithm is not compromised. The worst that can

happen is that a node is expanded which would have been pruned had the dequeuing order been different. This observation suggests that a priority queue with *weaker coherence* is appropriate.

An implementation of the priority queue was developed which partitions the set of nodes between all representatives in the SADT. Each representative now holds a local priority queue ADT which services local *enqueue* and *dequeue* operations. To enable the algorithm to perform as well as possible, a local dequeue operation should deliver a node which is *approximately* the best node available in the system. To provide this probabilistic behaviour, a daemon thread at each representative periodically shuffles a number of high priority nodes from the local queue to another representative The performance of the resulting diffusion process was enhanced through the application of a simple threshold-based load-balancing heuristic.

The new implementation was directly substituted for the unitary implementation. The best performance of enqueue and dequeue operations improved by two orders of magnitude compared to the unitary implementation on a network of workstations. Whilst average dequeue performance is good, performance can suffer due to starvation if significant load imbalances exist.

### 5.2.3 Priority Queue termination detection

The accumulator `counter` used for termination detection poses a problem. It is updated often, and it must be sequentially consistent to solve the consensus problem involved in termination detection[27]. To improve the performance of the algorithm therefore, an alternative approach is required.

The sequential algorithm terminates when the priority queue becomes empty (figure 7a). In the parallel case a worker observing the queue to be empty is insufficient as it may be only transiently empty before another worker enqueues more work (figure 7b). This has led to the adoption of a *blocking* semantics for the dequeue operation.

In the parallel case, termination should occur when all workers are waiting for work that does not exist.

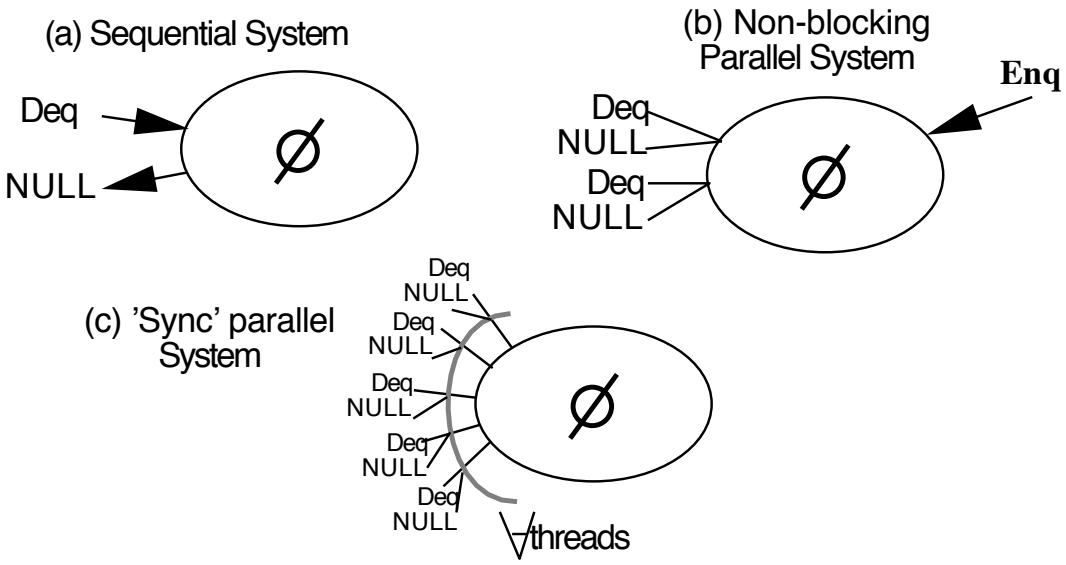


FIGURE 7: Termination detection on an empty priority queue.

This corresponds to the sequential code when the queue is empty and the only worker is attempting to dequeue. If the condition is detected when *all* parallel workers are attempting to dequeue and the queue is empty, then this can signal termination (figure 7c) as there is no possibility for new data to arise in the queue.

Termination detection is thus realised by implementing a protocol inside the priority queue[24]. In each address space, each worker thread is registered with the priority queue representative. When all representatives are empty and their locally registered workers are all blocked waiting to dequeue, then the dequeue operation returns `NULL` to all the workers across the system. This signals that the queue is empty and cannot become non-empty before the dequeue operation completes. The workers can thus safely use this return value to trigger termination. The accumulator `counter` can therefore be eliminated.

Detecting termination in distributed applications is a complex issue, often resulting in specialised solutions. This approach is an example of a general and re-usable mechanism, captured using the SADT model.

### 5.2.4 Optimised Performance

As a result of applying these three optimisations, a very significant improvement in the performance of the application was obtained. The reduction in overheads is clearly shown in figure 8. The percentage of

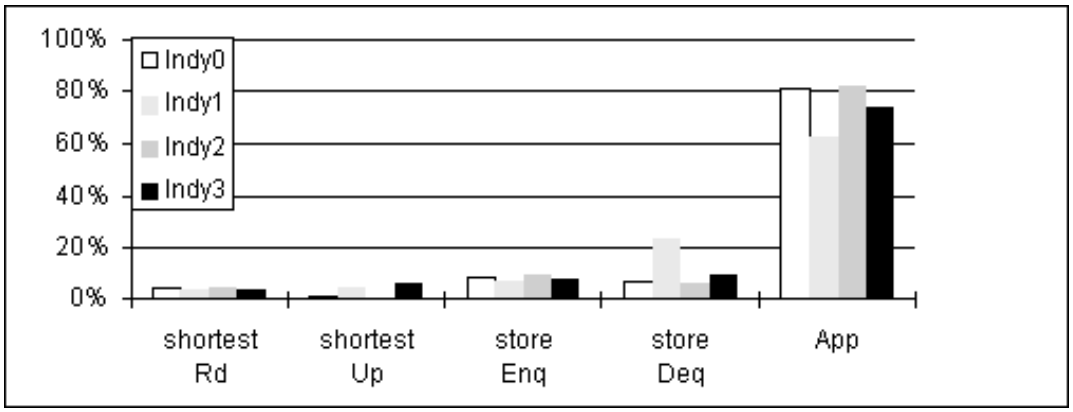


FIGURE 8: Execution time spent performing each SADT operation in TSP solver running the same example 20-city problem on a 4-processor network using the optimised SADTs `shortest` and `store`.

execution time available to the application now averages $75\%$ over all processors, delivering a speedup of $3$.

### 5.2.5 Cray T3D SADT implementations

The insights into the application identified for the network of workstations are equally applicable to the Cray T3D. A replicated accumulator is used, servicing read operations locally. A shuffling priority queue could have been implemented, but instead another partitioned style was chosen which *stripes* enqueue and dequeue requests across individual priority queue elements on each processor[34]. This implementation makes use of the same coherence optimisation exploited by the shuffling priority queue.

Two counters are used to implement the priority queue, to select the target processor on which to perform the enqueue/dequeue operation. Locks are then acquired on this processor, and synchronisation is performed to wake any blocked dequeuing process when an enqueue occurs locally.

The priority queue SADT implemented is not fully coherent. Instead, it delivers one of the $N$ highest priority elements in the system to a dequeuer, where $N$ is the number of processors. The priority queue SADT additionally implements the termination detection mechanism identified in section 5.2.3. The implementation therefore makes use of the same observation about the behaviour of the application used to drive the network of workstations priority queue implementation.

## 5.3 Performance Results

Figure 9 shows the execution time and speedup results for the application running on both the network of workstations and Cray T3D platforms. In both cases, the parallel application achieves high performance, and exhibits an impressive degree of scalability.

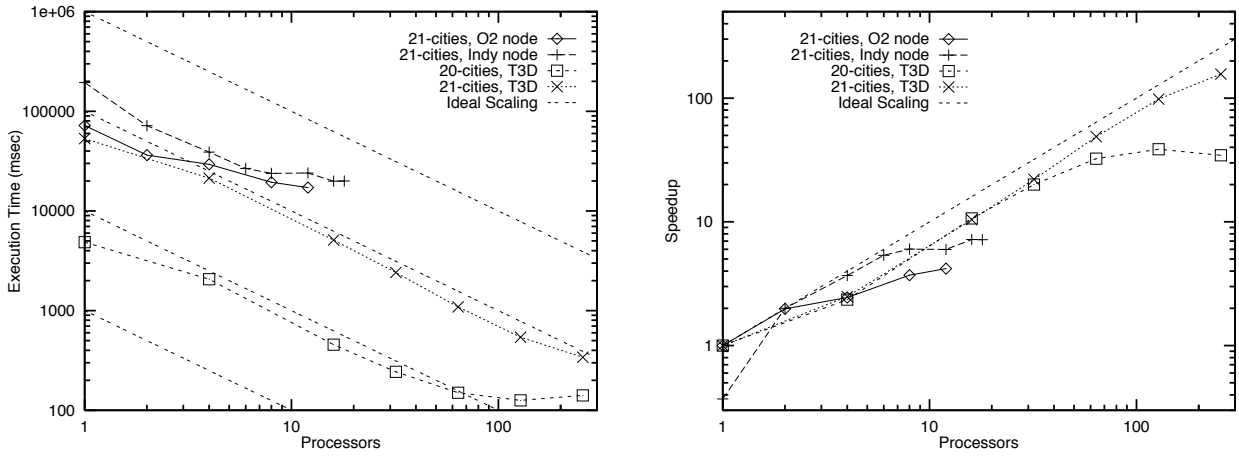


FIGURE 9: Execution time (left) and speedup (right) for the TSP solver on example problems running on a network of O2 and Indy workstations, and the Cray T3D.

The execution time curves for the network of workstations systems starts to deviate significantly from the ideal approaching 10 processors. This is due to the ethernet connecting the workstations beginning to saturate for this size of system, limiting scalability. Two networks of workstations were used, a network of R4000-based SGI Indy machines, and a network of R5000-based SGI O2 machines. The raw performance of the O2 is roughly twice that of the Indy, but with the same communication speed. This imbalance ex-

plains the poorer scaling behaviour of the O2 experiments. Also, the O2 machines share a single ethernet strand, whereas the Indy network is split up by network bridges, allowing it to sustain a higher aggregate bandwidth. The speedup achieved by the Indy network is higher across the range of machine sizes, peaking at $7.21$ on $16$ workstations. The highest speedup for the O2 network was $4.19$ on $12$ workstations. The speedup characteristic for the Indy network is normalised at $2.0$ on $2$ workstations. On one workstation, the local physical memory was exceeded during execution, causing a large amount of swapping to occur. On two workstations, the partitioning of the priority queue avoided this problem.

The execution time and speedup results for the T3D application display a high degree of performance and scalability. The single-node application performance of the T3D is slightly in excess of that achieved by an O2. The T3D performance scales for both problem sizes almost linearly to $100$ processors. Beyond this size, the execution time for the smaller problem starts to become limited by the application start-up overheads. The performance achieved for the larger problem however scales well up to $256$ processors, achieving a peak speedup of $156.5$. The overheads degrading machine performance could be addressed by a weaker implementation of the priority queue SADT, reducing the demands on the underlying network fabric.

## 5.4 Benefits of the SADT approach

The high-level coding of the TSP application has hidden the details of parallel execution from the application programmer. To enable performance to be enhanced, these details are exposed in a limited way, through choice over implementations to suit particular usage patterns and to serve particular coherence requirements. This has dramatically simplified the view of the application programmer compared with comparable programming efforts[22].

The optimisations afforded by the use of type information, usage pattern and coherence have allowed es-

sentially the same high-level application code, expressing a fine-grained algorithm, to be implemented efficiently on two diverse parallel platforms. The application achieves impressive levels of performance and scalability on both.

# 6 Conclusions

Typed shared memory systems are an important approach to the high-level and efficient implementation of irregular applications. Many proposed systems suffer from:

- representation mis-match, leading to poor performance where applications use types in ways inappropriate (or even pathological) to the underlying implementation;
- over-strict coherence, leading to inefficiencies from excessive coherence-maintenance network traffic; and
- increased programming complexity arising from the need for an unfamiliar programming style in the definition of application-level types.

These problems have been addressed by developing a system of Shared Abstract Data-types. The programming complexity issue is tackled by minimising the set of types required to a general-purpose (and thus highly re-usable) core, amortising the development effort over repeated uses. Representation mismatch is addressed by providing multiple implementations of types, suited to different patterns of usage. Coherence maintenance traffic is minimised by allowing substitution of SADTs with weaker coherence guarantees where appropriate, providing a mixed coherence model.

The benefits of this approach have been demonstrated through the implementation of a fine-grained parallel application. Using prototype SADT systems, essentially the same application code has been run on

two very different parallel platforms. Through selection of the underlying types with insight into the requirements of the algorithm, high and scalable performance and application code portability have been demonstrated.

## 6.1 Further work

Two major issues for further development of SADT implementations lie in the specification of both usage and coherence properties. Currently, the choice of implementation lies with the application programmer. Ongoing work is focussing on specifying coherence properties at the interface, delivering a true abstract type system, thus allowing application code to be verified correct using a more formal approach. This is seen as an important step towards the understanding of the interactions between type systems, concurrency and distribution.

Work is ongoing toward providing a firm theoretical foundation and semantics for SADTs, and a basis for optimising transformations and costs. The styles of sharing evident in real applications are being studied to extract, specify, and characterise a set of frequently useful SADTs, and to identify a taxonomy of sharing, with the goal of achieving a full set that is sufficiently rich for practical applications.

Costing SADT implementations using the WPRAM approach provides a way of reasoning about the performance benefits of different implementation styles. Initial work on queues has suggested that this approach is both feasible and generates useful results[36]. Ongoing work is applying this methodology to futher types in the T3D SADT library.

The SADT approach has proved to be capable of delivering both a high level of abstraction and high, scalable performance to the parallel application programmer. Experience will be extended in the future with further application study and the development of the SADT systems and library, as a basis for highly portable, high-level, high-performance implementations of scalable parallel applications.

# References


[1] Gul A. Agha. *Actors.* MIT Press, 1986.

[2] Andreas Müller and Roland Rühl. Extending High Performance Fortran for the support of unstructured computations. In *9th ACM Internetaion Conference on Supercomputing*, July 1995.

[3] M.S. Atkins and M.Y. Coady. Adaptable Concurrency Control for Atomic Data-types. *ACM Transactions on Computer Systems*, 10(3):190–225, August 1992.

[4] Hagit Attiya and Roy Friedman. Limitations of fast consistency conditions for distributed shared memories. *Information Processing Letters*, 57:243–248, 1996.

[5] Hagit Attiya and Jennifer L. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[6] Henri Bal, Andrew S. Tannenbaum, and M. Frans Kaashoek. Orca: a language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[7] J.K. Bennet, J.B. Carter, and W. Zwaenpoel. Munin: distributed shared memory based on type-specific memory coherence. *ACM SIGPLAN Notices*, 25(3):168–176, March 1990.

[8] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical Report CMU-CS-93-119, School of Computer Science, Carnegie-Mellon University, March 1993.

[9] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. Technical Report 115, DEC Systems Research Center, December 1995.

[10] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA'86*, pages 78–86. ACM Press, September 1986.

[11] S. Booth, J. Fisher, N. MacDonald, P. Maccallum, E. Minty, and A. Simpson. Parallel Programming on the Cray T3D, Version 1.1. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, August 1994.

[12] Luca Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center (SRC), Palo Alto, 1993.

[13] Soumen Chakrabarti and Katherine Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *ACM Symposium on Principles and Practice of Parallel Programming*, June 1993.

[14] John Chandy, Steven Parkes, and Prithviraj Banerjee. Distributed object oriented data structures and algorithms for VLSI CAD. In A. Ferreira, J. Rolim, Y. Saad, and T. Yang, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume LNCS 1117, pages 147–158. Springer-Verlag, 1996.

[15] A.A. Chien and W.J. Dally. Concurrent Aggregates. *ACM SIGPLAN Notices*, 25(3):187–196, March 1990.

[16] Andrew A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[17] Simon Dobson and Andy Wellings. A system for building scalable parallel applications. In Nigel Topham, Roland Ibbett, and Thomas Bemmerl, editors, *Programming environments for parallel computing*, pages 218–230. North-Holland Elsevier, 1992.

[18] Emin Gün Sier and Stefan Savage and Przemyslaw Pardyak and Greg DeFouw and Mary Ann Alapat and Brian Bershad. Writing an operating system with Modula-3. In *Workshop on Compiler Support for System Software*, February 1996.

[19] O-J. Dahl et al. Simula 67. 1967.

[20] Michael J. Feely and Henry M. Levy. Distributed Shared Memory with Versioned Objects. In *OOPSLA'92*, pages 247–262. ACM Press, 1992.

[21] High Performance Fortran Forum. High Performance Fortran language specification version 1.1. http://www.vcpc.univie.ac.at/mirror/HPFF/hpf1/hpf-v11/hpf-report.html, November 1994.

[22] Bernard Gendron and Teodor Gabriel Crainic. Parallel Branch-and-Bound algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, December 1994.

[23] D.M. Goodeve, J.R. Davy, P.M. Dew, and J.M. Nash. The Consistency Properties of a Scalable Concurrent Queue. Technical Report 96.35, School of Computer Studies, University of Leeds, UK, December 1996.

[24] D.M. Goodeve, C. Tofts, and S.A. Dobson. ThreadGroups: A lightweight distributed co-ordination mechanism. Technical Report 97.07, School of Computer Studies, University of Leeds, UK, March 1997.

[25] Don Goodeve, John Davy, and Chris Wadsworth. Shared Accumulators. In *Transputer Applications and Systems '95*, pages 518–528. IOS Press, 1995.

[26] C.L. Hartley and V.S. Sunderam. Concurrent Programming with Shared Objects in Networked environments. *IEEE Transactions*, pages 471–478, 1993.

[27] Maurice Herlihy. Wait-Free Synchronisation. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[28] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[29] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[30] B.H. Liskov, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Comm. ACM*, 20(8), August 1977.

[31] John D.C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Ka rel. An Algorithm for the Travelling Salesman Problem. *Operations Research*, 11:972–989, 1963.

[32] W. F. McColl. An Architecture Independent Programming Model For Scalable Parallel Computing. In J. Ferrante and A. J. G. Hey, editors, *Portability and Performance for Parallel Processing*. John Wiley and Sons, 1993.

[33] J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronisation on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[34] 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.

[35] J. M. Nash, M. E. Dyer, and P. M. Dew. Designing Practical Parallel Algorithms for Scalable Message Passing Machines. In *WTC'95 World Transputer Congress*, pages 529–544, 1995.

[36] J.M. Nash, P.M. Dew, and J.R. Davy. A Parallelisation Approach for Supporting Scalable and Portable Computing. Submitted to Europar'97, Passau, Germany., 1997.

[37] Greg Nelson. *Systems Programming with Modula-3*. Prentice-Hall series in Innovative Computer Science. Prentice-Hall, 1991.

[38] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24:52–60, 1991.

[39] Steven Parkes, John A. Chandy, and Prithviraj Banerjee. A library-based approach to portable, parallel, object-oriented programming: interface, implementation and application. In *Supercomputing 94*, pages 69–78, 1994.

[40] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.

[41] Nir Shavit and Dan Touitou. Elimination Trees and the Construction of Pools and Stacks. In *7th Annual Symposium on Parallel Algorithms and Architectures*, pages 54–63. ACM Press, April 26th, 1995.

[42] Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Katherine Yelick. Support for Portable Distributed Data Structures. In *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1995.

[43] Greg Wilson and Henri Bal. Using the Cowichan problems to assess the usability of Orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, 1996.

[44] W. Wulf, R. London, and M. Shaw. An Introduction to the construction and verification of Alphard programs. *IEEE Trans. Soft. Eng.*, 2(4), 1976.