JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **49,** 156–167 (1998)
ARTICLE NO. PC981427

Toward a Model for Shared Data Abstraction with Performance[1]

D. M. Goodeve

Advanced Computer Architectures Group, Department of Computer Science, University of York, York YO1 5DD, United Kingdom

S. A. Dobson

Distributed Systems Group, Department of Computer Science, Trinity College, Dublin, Ireland

J. M. Nash, J. R. Davy, P. M. Dew, M. Kara

Scalable Systems and Algorithms Group, School of Computer Studies, University of Leeds, Leeds LS2 9JT, United Kingdom

and

C. P. Wadsworth [2]

Independent Consultant

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 synchronization. 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 task-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. © 1998 Academic Press

[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]Wadsworth was formerly at the CLRC Rutherford Appleton Laboratory, Didcot, Oxfordshire, UK.

1. INTRODUCTION

Writing codes for parallel machines involves excessive effort compared with sequential programming. To simplify program development, abstraction is essential to enable clean separation of application code from the *assembly-level mechanisms* coordinating parallel execution, thus encapsulating the issues of communication, synchronization, and distribution. The goal is 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 High-Performance Fortran (HPF). Irregular applications however pose further problems and are a largely unsolved area [1].

This paper presents an effective approach to high-performance parallel abstraction, based on shared abstract data-types (SADTs). 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 optimization routes.

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 *et al.*’s Orca language [4, 26], Chien and Dally’s Concurrent Aggregates language [12], Chandy *et al.*’s ProperCAD library system [11], and the MULTIPOL project of Yelick *et al.* [10, 25]. 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 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 amortized over reuse of SADTs between applications. Although the creation of SADTs is relatively complex, the final implementations may encapsulate a wide range of variously applicable optimizations including design idioms, weak coherence, task-specific representations, and latency hiding. 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.

Section 2 briefly introduces existing typed shared memory systems and contrasts these with the SADT approach. The design of prototype SADT systems for both a network of workstations and a massively parallel machine are then discussed in Section 3. The behavior of these prototypes is investigated in Section 4, where they are used to support the same parallel Branch-and-Bound solver application for the travelling salesman problem (TSP). Results are presented for both systems, demonstrating both the level of abstraction achieved and the performance delivered. Finally in Section 5, conclusions are drawn about the SADT approach and its future development.

2. TYPED SHARED MEMORY SYSTEMS

Types form the basis of much of modern software engineering in sequential computing [9], decoupling the layers of a software system, simplifying interactions, and allowing complexity to be controlled. This section examines several approaches to extending types into distributed concurrent environments.

2.1. Type Sharing

The simplest form of shared type makes a sequential object implemented at some node accessible remotely by processes at other nodes. Concurrency control typically makes each object a monitor to maintain data coherence. A heavily used object can suffer from severe contention, both as a network hot-spot and through locking.

The Orca language [4] attempts to avoid contention in special cases by adding a replicated implementation style. Read operations occur against a local replica; update operations use (very expensive) ordered atomic broadcasts to update all replicas. This is an important optimization for objects which are updated very infrequently. Network traffic may be reduced by migrating an object closer to its caller, an approach adopted in the Emerald [7] and DoPVM [18] systems. While avoiding the false-sharing phenomenon of low-level distributed shared memory (DSM) systems [23], performance can still be compromised by excessive object movement.

Higher performance may be achieved through focused design effort on specific data structures and algorithms, e.g., [24]. A common implementation style represents a single shared abstraction through a distributed community of intercommunicating components [10–13, 25]. Clients access the abstraction though a local component; cooperation within the community supports the overall abstraction. This permits a number of important techniques to be deployed behind the type abstraction barrier.

2.2. Representation Mismatch

Representation mismatch is the situation where (possibly hidden) implementation choices interact unfavorably with a type's mode of use. This is not a failure of type abstraction per se, but rather a consequence of the costs of different operations under different representations. An example from the sequential world is a singly linked list, where iterating in one direction has a wildly different cost to iterating in the other. This situation is exacerbated by distributed processing, where mismatch may lead to vastly increased network-induced delays. There is typically no single representation which is well matched to every pattern of use, and mismatch becomes especially severe when only a small repertoire of distributed implementation schemes are available [26].

2.3. Coherence

Coherence concerns the views that different processes have of a single data item. Strict sequential consistency becomes prohibitively expensive as network latency increases [3]. If coherence can be relaxed, it follows that overheads incurred in maintaining the shared data abstraction may be reduced [5, 14].

At the present state of the art, recognizing that weak coherence can be applied is largely down to observation of the algorithms involved. Earlier studies [10] have demonstrated that coherence requirements can be extracted for particular algorithms. In general algorithms cannot tolerate the relaxation of coherence in arbitrary variables. Conversely, implementing strong versions of types in terms of weaker versions is too complex to be deferred to the application programmer. This suggests the provision of a mixture of coherence models for each type in a library [2]. Careful design is needed to ensure that such weakened types provide the programmer with useful and understandable abstractions.

2.4. Shared Abstract Data-Types

The SADT model synthesizes existing experiences with multiple representations and weak coherence into a small, extensible set of types covering the common sharing modes encountered in parallel algorithms. By maintaining a small set of types, the complex low-level implementation effort is recouped over repeated reuse. An application is coded against the abstract interfaces from the different SADTs. The actual implementation of each individual instance may be specified at compile-time or may use run-time information. This allows optimized implementations to be deployed progressively without affecting the application-level code.

3. PROTOTYPE SADT IMPLEMENTATIONS

The main prototyping environment used in this work has been developed on an Ethernet-connected network of workstations. This type of system is challenging for parallel programming due to high communication overheads, requiring highly effective use of type, usage, and consistency information to drive efficient SADT implementations.

The SADTs written for this platform rely on a simple underlying run-time executive. The executive, SADTs, and applications are written in Modula-3 [22]. 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. Interaction between address spaces is facilitated by the Network Objects system [6].

An SADT is composed of a set of representatives, one per address space. The application code in each address space communicates with its local representative using a simple front-end object, embedding SADTs into the existing type framework. This is similar to the implementation style of concurrent aggregate systems [11, 12] except that the local representative is always invoked, thus avoiding any unnecessary network traffic.

The representatives act as a substrate for coordinating activity across address spaces, permitting many implementation styles as shown in Fig. 1. The simplest *unitary* SADT implementation involves all requests being forwarded to an object implementing the abstraction in a single address space. While it is easy to realize, performance is bounded by both the network and the performance of the processor hosting the actual object.

The second style is a replicated object, kept coherent using an ordered write-update broadcast protocol, as in Orca [4]. Each of these two 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, to be applied *en masse* either through explicit synchronization or periodically in the background.

In the most general, *partitioned* style, the representation of the SADT is spread across address spaces; representatives contain part of a complete data structure, organizing the distribution of this structure transparently to the user. This third style gives the

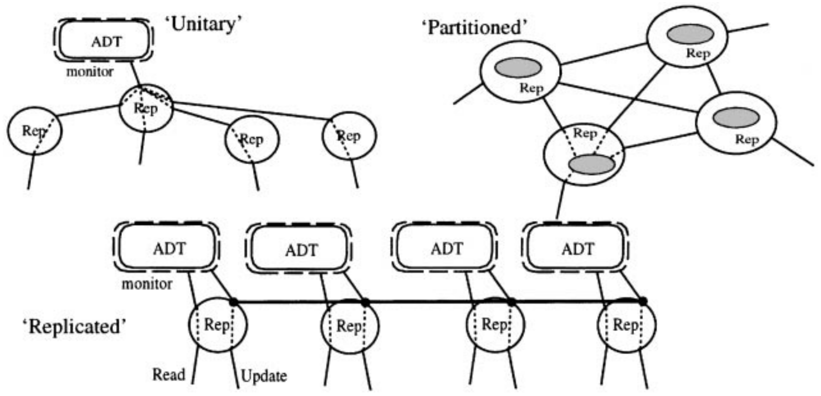

FIG. 1. Three different implementations styles for SADTs using the representative architecture; unitary (centralized), replicated, and partitioned.

implementer the most freedom to make use of individual type, usage, and coherence information in design and is open to many possible implementation techniques.

3.1. Implementing SADTs Using Shared Memory Primitives

The prototype SADT system on the Cray T3D is coded in C, using the common *object-oriented C* style of programming. The SADT system is constructed using the primitive operations of the WPRAM computational model [21] and the underlying Cray SHMEM shared memory primitives library [8].

The T3D SHMEM library allows processor-local memories across the machine to be addressed from any other processor without interrupting the target and supports a small repertoire of distributed atomic operations. These primitives make it straightforward to implement the synchronization structures required by the WPRAM and for the implementation of SADTs.

The prototype system statically binds the required SADTs into an SPMD program. This allows the SADT implementations to find the required coordination data structures at the same address on all processors (Fig. 2).

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. It is proposed that a distributed reference-counting collector is implemented in a future system.

4. EXAMPLE BRANCH-AND-BOUND APPLICATION

A parallel solver for the TSP was developed from a fine-grained sequential code implementing the branch-and-bound algorithm of Little *et al.* [20]. The algorithm proceeds by refining a description of the solution space as a tree of *nodes*. Each node represents a set of feasible tours, with the root of the tree representing all feasible tours. The expansion of a node into its children is guided by a heuristic which makes it likely that one child contains the optimum tour and conversely that the other child does not. A second heuristic directs the choice of the likely node to contain the optimal solution at the next execution step. The heuristic guides the algorithm to next expand the node with the lowest length lower-bound of all those available. This is achieved through the use of

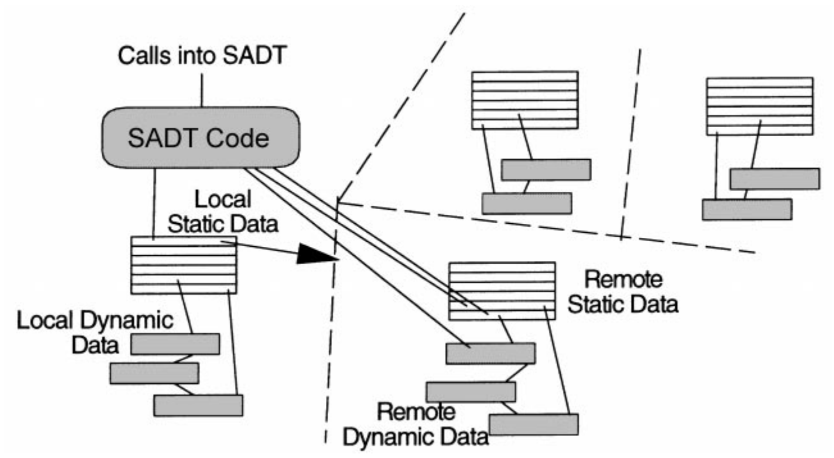

FIG. 2. Illustration of and SADT implementation on the Cray T3D using the SHMEM substrate. An access to the local static data table indicates the processor at which the following operation should be performed. Remote dynamic data is accessed by indirecting through the remote static data table.

a *priority queue* to store the nodes of the tree, where the maximal priority corresponds to the lowest node lower-bound. Leaf nodes, representing single tours, are used when found to *improve* an upper-bound on the length of the optimal solution and drive the pruning of the search space.

The parallel application consists of a number of identical worker processes running asynchronously, only cooperating through the use of SADTs as outlined by the pseudo-code in Fig. 3.

Two SADTs are used: a priority queue `store` storing the current set of nodes and an *accumulator* SADT `shortest`, an abstraction of a variable with an update function [16], storing the best known tour. The accumulator can be read to return the current value or updated in which case its update function (in this case a minimum function) is applied to the current value and the argument of the update operation. By combining the tour with

```
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;
    }
  }
}
```

FIG. 3. 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.

its length as the accumulator stored type, the best known solution is also retained on a successful update. The contribution of this solver over others proposed in the literature [15] is the abstraction from the low-level details of data management that it achieves, cleanly expressing the core algorithm.

4.1. Optimizations

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 (Fig. 1). An execution profile of the application (Fig. 4) clearly demonstrates that the vast majority of the execution time is spent on SADT operations, owing to the high, synchronous network traffic generated, leaving only a few percent for the application code. A common route to improving performance in this situation is to increase the computation grain-size, reducing the dominant effect of network delays, however obscuring the application algorithm. To avoid this, usage patterns and coherence requirements were exploited to optimize inside the SADT abstraction boundary.

A study of the TSP code reveals that it requires only basic semantic guarantees from the SADTs to ensure correctness. The priority queue need only behave as an unordered lossless store. The accumulator must not return a value lower than has been written to it. Harder semantic guarantees such as strict, coherent priority and a fully coherent accumulator may be regarded as properties that can be traded off against improved implementation efficiency. The balance of these tradeoffs lies in the sensitivity of the algorithm to weakened guarantees above the minimal level required for correctness. It is hypothesized that this type of tradeoff is general and can be exploited through the SADT approach.

Each time around the main loop of the algorithm, the accumulator `shortest` is read. It is only updated however when a complete solution is found. A *replicated* accumulator implementation (Fig. 1) exploits this usage pattern. Cached copies of the accumulator value are updated using a write-update protocol. This implementation improves read latency by two orders of magnitude, while degrading write latency by one order, which matches well to the application usage pattern.

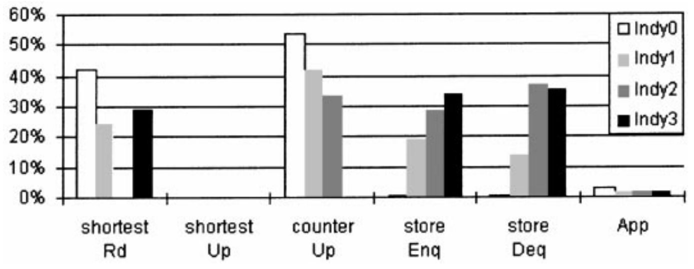

FIG. 4. Execution time spent performing each SADT operation, and the remaining time for executing the application for an example 20-city problem solved on a network of 4 SGI-Indy workstations connected using TCP/IP over a 10MBit/sec Ethernet.

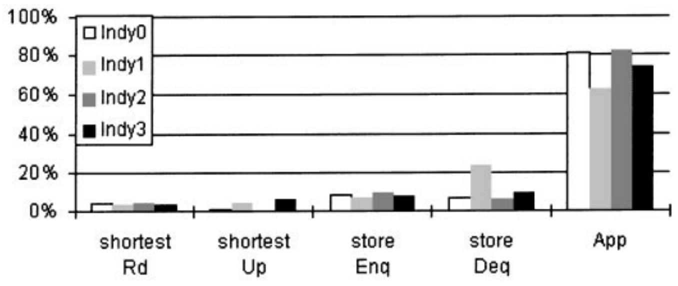

FIG. 5. Execution time spent performing each SADT operation in TSP solver running the same example 20-city problem on a 4-processor network using the optimized SADTs `shortest` and `store`.

To make priority queue operations fast, they must be serviced locally. An implementation of the priority queue was developed which partitions the set of nodes between all SADT representatives. Each representative now holds a local priority queue ADT which services local *enqueue* and *dequeue* operations. A local dequeue operation should deliver a node which is *approximately* the best node available in the system. 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 best performance of enqueue and dequeue operations improved by two orders of magnitude compared to the unitary implementation on a network of workstations. While average dequeue performance is good, latency can suffer due to starvation, especially due to poor load balance toward the end of a run. The shuffling priority queue provides an example of how the irregular-application load balancing problem can be tackled within the SADT model.

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 [19]. A simple blocking or nonblocking semantics for the queue is inadequate for this task without some external protocol. A general mechanism was identified which allowed the required protocol to be implemented in a lightweight form *inside* the priority queue. Knowing how many threads are accessing it, the queue can signal to all threads simultaneously when they are all blocked waiting, equivalent to the state that the distributed structure is exhausted [17].

The reduction in overheads arising from these optimizations is clearly shown in Fig. 5. The percentage of execution time available to the application now averages 75% over all four processors, delivering a speedup of 3.

The same optimizations and tradeoffs are also exploited on the T3D implementation, including a replicated accumulator. Another partitioned style of priority queue was implemented which *stripes* enqueue and dequeue requests across individual priority queue elements on each processor [21]. This is more suited to the T3D programming model. The same application code was used as on the network of workstations, albeit implemented in C.

4.2. Performance Results

A peak speedup of 7.21 was achieved on 16 SGI-Indy machines, limited beyond this point by network saturation. The O2 networks deliver worse scalability due to a lower

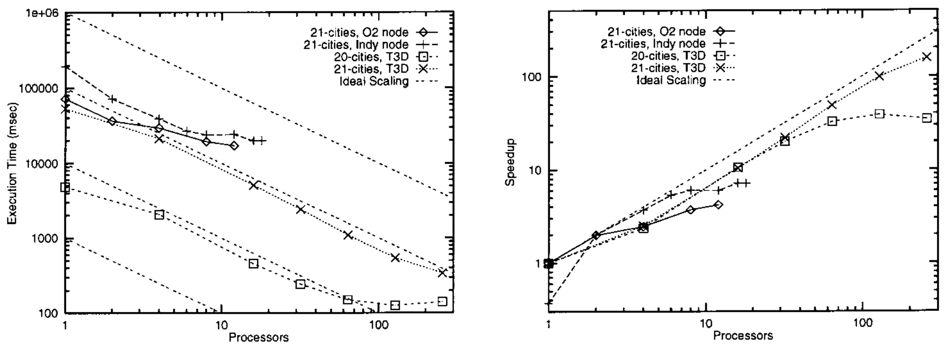

FIG. 6. Execution time (left) and speedup (right) for the TSP solver on example problems running on a network of O2 and Indy workstations, and a Cray T3D.

communication/computation ratio. The speedup characteristic for the Indy network is normalized at 2.0 on two workstations as on one workstation, local physical memory was exceeded during execution, causing massive overheads.

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, due to the available low-overhead scalable noninterrupting hardware communications and synchronization. Start-up overheads dominate the small problem execution time beyond this scale, however the speedup achieved for the larger problem scales well up to 256 processors, achieving a peak speedup of 156.5 (see Fig. 6).

6. CONCLUSIONS

Many proposed typed shared memory systems suffer from representation mismatch, 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.

In the SADT model, programming complexity is tackled by minimizing the set of types required to a general-purpose (and thus highly reusable) core, amortizing the development effort over repeated use. Representation mismatch is addressed by providing multiple implementations of types, suited to different patterns of usage. Coherence maintenance traffic is minimized by allowing substitution of SADTs with weaker coherence guarantees where appropriate, providing a mixed coherence model.

Essentially the same application code has been run on two very different parallel platforms. Selection of the underlying types with insight into the requirements of the algorithm has delivered high and scalable performance, and demonstrated portability.

Ongoing work is focused on specifying coherence requirements and usage properties at the interface, delivering a true abstract type system. This is seen as an important step toward the understanding of the interactions between type systems, concurrency, and distribution. The styles of sharing evident in real applications are being studied to extract, specify, and characterize a set of frequently useful SADTs, sufficiently rich for practical applications. Work is also ongoing toward providing a firm theoretical foundation and semantics for SADTs, and a basis for optimizing transformations and costs. Further application study and the development of the SADT systems and library appear to provide a very promising basis for highly portable, high-level, high-performance implementations of scalable parallel applications.

REFERENCES

1. A. Müller and R. Rühl, Extending high performance Fortran for the support of unstructured computations, *in* "9th ACM International Conference on Supercomputing," July 1995.
2. M. S. Atkins and M. Y. Coady, Adaptable concurrency control for atomic data-types, *ACM Trans. Comput. Systems* **10,** 3 (August 1992), 190–225.
3. H. Attiya and R. Friedman, Limitations of fast consistency conditions for distributed shared memories, *Inform. Process. Lett.* **57** (1996), 243–248.
4. H. Bal, A. S. Tannenbaum, and M. F. Kaashoek, Orca: A language for distributed programming, *ACM SIGPLAN Notices* **25,** 5 (May 1990), 17–24.
5. J. K. Bennet, J. B. Carter, and W. Zwaenpoel, Munin: Distributed shared memory based on type-specific memory coherence, *ACM SIGPLAN Notices* **25,** 3 (March 1990), 168–176.
6. A. Birrell, G. Nelson, S. Owicki, and E. Wobber, "Network Objects," Technical Report 115, DEC Systems Research Center, December 1995.
7. A. Black, N. Hutchinson, E. Jul, and H. Levy, Object structure in the emerald system, *in* "OOPSLA'86," pp. 78–86, Assoc. Comput. Mach. Press, New York, September 1986.
8. 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, Edinburgh EH9 3JZ, August 1994.
9. L. Cardelli, "Typeful programming," Technical Report 45, Digital Systems Research Center (SRC), Palo Alto, 1993.
10. S. Chakrabarti and K. Yelick, Implementing an irregular application on a distributed memory multiprocessor, *in* "ACM Symposium on Principles and Practice of Parallel Programming," June 1993.
11. J. Chandy, S. Parkes, and P. Banerjee, Distributed object oriented data structures and algorithms for VLSI CAD, *in* "Parallel Algorithms for Irregularly Structured Problems," (A. Ferreira, J. Rolim, Y. Saad, and T. Yang, Eds.), Lecture Notes in Computer Science, Vol. 1117, pp. 147–158, Springer-Verlag, Berlin/New York, 1996.
12. A. A. Chien, "Concurrent Aggregates," MIT Press, Cambridge, MA, 1993.
13. S. Dobson and A. Wellings, A system for building scalable parallel applications, *in* "Programming Environments for Parallel Computing," (N. Topham, R. Ibbett, and T. Bemmerl, Eds.), pp. 218–230, North-Holland Elsevier, Amsterdam, 1992.
14. M. J. Feely and H. M. Levy, Distributed shared memory with versioned objects, *in* "OOPSLA'92," pp. 247–262, Assoc. Comput. Mach. Press, New York, 1992.
15. B. Gendron and T. G. Crainic, Parallel branch-and-bound algorithms: Survey and synthesis, *Oper. Res.* **42,** 6 (December 1994), 1042–1066.
16. D. Goodeve, J. Davy, and C. Wadsworth, Shared accumulators, *in* "Transputer Applications and Systems '95," pp. 518–528, IOS Press, 1995.
17. D. Goodeve, C. Tofts, and S. Dobson, "Lightweight Distributed Termination Detection using Thread-groups, submitted.
18. C. L. Hartley and V. S. Sunderam, Concurrent programming with shared objects in networked environments, *IEEE Transactions*, pp. 471–478, 1993.
19. M. Herlihy, Wait-free synchronisation, *ACM Trans. Program. Languages Systems* **11,** 1 (January 1991), 124–149.
20. J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Korel, An algorithm for the travelling salesman problem, *Oper. Res.* **11** (1963), 972–989.

21. J. M. Nash, P. M. Dew, and M. E. Dyer, A scalable concurrent queue on a message passing machine, *Comput. J.* **39,** 6 (1996), 483–495.

22. G. Nelson, “Systems Programming with Modula-3,” Prentice-Hall series in Innovative Computer Science, Prentice-Hall, New York, 1991.

23. B. Nitzberg and V. Lo, Distributed shared memory: A survey of issues and algorithms, *IEEE Comput.* **24** (1991), 52–60.

24. N. Shavit and D. Touitou, Elimination trees and the construction of pools and stacks, *in* “7th Annual Symposium on Parallel Algorithms and Architectures,” pp. 54–63, Assoc. Comput. Mach. Press, New York, April 26, 1995.

25. C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, Support for portable distributed data structures, *in* “Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers,” May 1995.

26. G. Wilson and H. Bal, Using the Cowichan problems to assess the usability of Orca, *IEEE Parallel Distrib. Tech.* **4,** 3 (1996), 36–44.

D. M. GOODEVE obtained the M.Eng. in electronic systems engineering from the University of York, UK in 1990, continuing to obtain his D.Phil. in 1994 for analysis work on VLSI-switch based communications fabric. After working on scalable abstract techniques as part of the TallShiP project in the School of Computer Studies at the University of Leeds, he took up his current lecturing post in the Advanced Computer Architectures Group of the Department of Computer Science at the University of York. His main research interests include scalable data abstraction techniques, coherence issues, runtime system design, and hardware acceleration.

S. A. DOBSON received the B.Sc. from the University of Newcastle upon Tyne in 1989 and a D.Phil. from the University of York in 1993 (with a thesis on programming environments for scalable parallel computers). He joined the Rutherford Appleton Laboratory in 1992, where he worked as a senior research fellow attached to the Well-Founded Systems Unit. In December 1997 he took up his present position as a research fellow in the Department of Computer Science, Trinity College, Dublin. His main research interests are in novel approaches to programming for distributed, and open and hostile environments, in particular through the use of extended type systems. He is also involved in the integration of programming languages with structured information sources such as databases and hyper-media.

J. M. NASH received the Ph.D. from Leeds University in computer science in 1993, and is currently employed as a research fellow in the School of Computer Studies. His interests include the design and analysis of scalable and portable algorithms, parallel models, and architectures. He has developed the WPRAM model, which extends the bulk synchronous parallelism paradigm to efficiently support irregular forms of parallelism. Recent work has focused on supporting shared abstract data-types on the Cray T3D, using the WPRAM to guide algorithm design and analysis. Initial results demonstrate the very high and scalable performance which this approach can support, in the areas of dynamic load balancing and irregular computations.

J. R. DAVY received his B.A. degree in mathematics at the University of Oxford in 1970, his M.Sc. in computer science at the University of Manchester in 1987, and his Ph.D. in computer science at the University of Leeds in 1993. He is currently a senior lecturer in computer science in the School of Computer Studies, University of Leeds. His current research interests include high-level approaches to parallel programming and the use of parallel computers to solve large-scale geographical problems.

P. M. DEW leads the Distributed Multi-Media Academic group (including Scalable Systems and Algorithms). In addition, he is Deputy Director of the Keyworth Institute for Manufacturing and Information Systems Engineering, leading the strategic program on virtual prototyping. He has published widely and is on the editorial board of the *Journal of Parallel Algorithms and Applications*. His research interests cover constraint solid modeling, scalable systems, and algorithms and is undertaking systems architecture research for distributed multi-media systems, to support virtual working systems. He is a current member of the EPSRC Computing College for Systems Architecture.

M. KARA received the Ph.D. in distributed computing systems from the University of Leeds in 1991, in the area of performance evaluation of dynamic load balancing systems. He is currently a lecturer in the School of Computer Studies at the University of Leeds. His research interests are in distributed multi-media systems and resource scheduling over ATM networks.

C. P. WADSWORTH has worked in research in many areas of computing for 30 years at the Universities of Oxford, Syracuse, and Edinburgh, and since 1981 at the Rutherford Appleton Laboratory in the UK. He is now an independent consultant specializing in research and coordination of research in parallel and distributed computing for the UK EPSRC and others. His current research interests include high-level approaches to parallel programming, abstractions for sharing and coordination between multiprocessor activities, and techniques for detection and transformation of scope for parallelism in non-parallel (sequential) programs.

Received January 14, 1997; revised December 27, 1997; accepted January 5, 1998