Programming with shared data abstractions

Simon Dobson[1] and Don Goodeve[2]

[1] Well-Founded Systems Unit, CLRC Rutherford Appleton Laboratory, UK
[2] School of Computer Studies, University of Leeds, UK

Abstract. We present a programming system based around a set of highly concurrent, highly distributed data structures. These *shared abstract data types* offer a uniform interface onto a set of possible implementations, each optimised for particular patterns of use and more relaxed coherence conditions in the data. They allow applications written using a shared data model to approach the performance of message passing implementations through a process of pattern analysis and coherence relaxation. We describe the programming system with reference to a sample application solving the travelling salesman problem. Starting from a naïve direct implementation we show significant speed-ups on a network of workstations.

1 Introduction

Parallel computing offers the prospect of high application performance. This can indeed be realised, but at the price of huge development and maintenance costs arising from the complexity of managing inter-processor interaction and application code in the same framework. Ideally an application programmer should only deal with the details of the application problem, reducing complexity to an acceptable and manageable level. Some method for abstracting away from the complexities of inter-processor interaction is therefore required. This has been accomplished most successfully for algorithms having a regular form where static problem decomposition is possible. Producing efficient codes for irregular problems is a significant challenge, as effort must be devoted to the effective management of an evolving computation.

In this paper we present an approach based on *shared abstract data types* or SADTs – an abstract type that can be accessed by many distributed processes concurrently. An SADT insulates an application from many different implementations of the abstraction, which may efficiently support different patterns of use and coherence models. The selection of the representation which best suits the application's exact requirements offers the highest possible performance without compromising the type abstraction boundaries.

Section two introduces an motivating problem. Section three describes the SADT model and its realisation in a prototype programming environment. Section four shows how a simple sequential solver for our illustrative problem may be progressively refined into a high-performance scalable parallel application. Section five compares and contrasts SADTs with related work from the literature. Section six offers some observations and future directions.

2 The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a well-known combinatorial optimisation problem. Given a set of cities and the distances between them an algorithm must find the shortest "tour" that visits all cities exactly once. In principle the entire search tree must be explored if an optimal solution is to be found, making the problem NP-complete. Problems of this form are often encountered in the context of organisational scheduling[18]. There are a number of possible approaches to the solution of the problem which broadly fall under the headings of *stochastic* algorithms, such as simulated annealing[17], and *deterministic* algorithms. Deterministic algorithms have the advantage of guaranteeing that the optimal solution is found, but usually can only deal with modest problem sizes due to time and space constraints. Stochastic algorithms trade improved performance for sub-optimal solutions.

One of the best-known deterministic algorithms is due to Little[14]. This algorithm employs two heuristics to guide the search for a solution by refining a representation of the solution space as a search tree. A node in the tree represents a set of possible tours, with the root representing the set of all possible tours. A step of the algorithm involves selecting a node in the tree and choosing a path segment not contained in this set. The path segment is selected by a heuristic designed so that the set of tours including the selected segment is likely to contain the optimal tour, and conversely that the set of tours that excludes it is unlikely to contain the optimal tour. A step therefore expands a node and replaces it by its two children – one including and one excluding the selected path segment. A second heuristic directs the algorithm to work on the most promising node next. A lower-bound on the length of any tour represented by a node can be computed, and the node selected to be explored next is that with the lowest lower-bound. Once a complete tour has been found, its length can be used to bound future exploration.

Little's algorithm is of particular interest for two reasons:

1. it is a *fine-grained algorithm* – a step of the algorithm involves only a relatively short computation; and
2. it is *irregular* – the evolution of the data cannot be predicted statically, and hence a significant amount of dynamic data management is involved.

These make it a challenging case study for any novel programming environment.

3 Shared Abstract Data Types

We have been exploring the use of *pattern-optimised representations* and *weakened coherence models* to allow a distributed data structure to optimise both the way its data is stored and the degree of coherence between operations on different nodes. These optimisations may occur without changing the type signature of the structure, allowing progressive optimisation. We refer to types with these properties as *shared abstract data types*, or SADTs[6]. The programming model for SADTs is the common one of a collection of shared data objects accessed concurrently by a number of processes independent of relative location or internal representation.

We have implemented a prototype programming environment to test these ideas, using the Modula-3[16] language with network objects[3] to implement bag, accumulator, priority queue and graph SADTs. This has proved to be an excellent and flexible test-bed. (We have also demonstrated more restricted environment, intended for supercomputers, using C.) There were four main design goals:

1. *integration* of SADTs into the underlying programming model and type system;
2. *extensibility* of the environment to derive new SADTs from the existing set, and allow additional base SADTs to be generated where necessary;
3. *incremental development* of applications using progressive optimisation; and
4. *high performance* applications when fully optimised.

We regard these goals as being of equal importance in any serious programming system.

3.1 Architecture

The heart of the SADT architecture is a network of *representatives*, one per processor per SADT. Each representative stores some part of the SADTs contents, and co-operates with the other representatives to maintain the shared data abstraction. Interactions with and between representatives occur through *events*, representing the atomic units of activity supported over the SADT. Events may be created, mapped to actions on values, buffered for later application, and so forth: the way in which representatives handle events defines the behaviour of the SADT, and allows subtly different behaviours to be generated from the same set of events.

The programmer's view of this structure is a local instance of some *front-end* type – usually a container type such as a bag of integers. The front-end provides a set of methods through which the shared data may be accessed: internally each is translated into one or more events which are passed to the representative for the SADT on the local node. It is important to realise that the events of the SADT do not necessarily correspond exactly to the user-level operations.

3.2 Events and Event Processing

Each SADT defines its own set of events, capturing the basic behaviour of the type being represented. For example, a bag SADT might define events to add an element and remove an element. These events and their eventual mapping to actions characterise the SADT.

There is a small core set of events which are applicable across all SADTs. One may regard these core events as providing the essential services to support efficient shared type abstraction. Many are "higher order", in the sense of taking other events as parameters. The set includes:

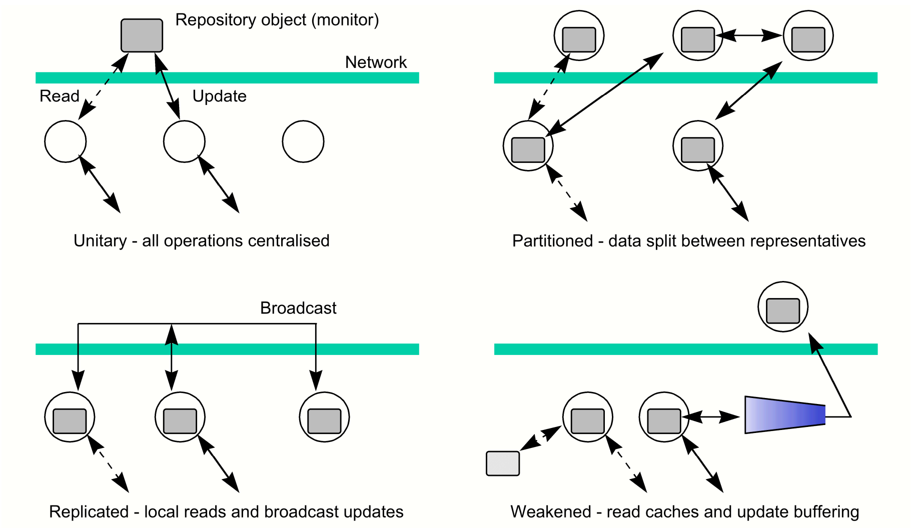
`LocalSync` – causes a representative to synchronise its view of the SADT with the other representatives. Typically this involves reading new values into local caches, or flushing pending events.
`Span(ev)` – performs event `ev` on all representative concurrently.
`Broadcast(ev)` – like `Span()`, but orders broadcasts at each representative (so all representatives will observe the same sequence of broadcasts).
`Bulk(ev1, ev2, …)` – submits a set of events in a single operation.

These events may be used to build higher-level operations. For example, an SADT may implement global synchronisation by issuing a `Span(LocalSync)` event[1], and fully replicated representations will often broadcast update events to all representatives. These events have efficient implementations based on concurrent spanning trees.

The critical importance of events is that they define "first-class" operation requests: representatives may manipulate events in complex ways before resolving them to an action. Event processing is at the core of the SADT model. Using the same set of events with the same mechanisms for rendering events into actions – the same type, in fact – an SADT may deploy several representations which differ in the exact way the events are buffered and combined. The different representations may efficiently support different coherence constraints and patterns in the use of the type.

[1] Or sometimes `Broadcast(LocalSync)`, but for many SADTs the extra ordering overhead is unnecessary.

There are a number of common processes which may be used to manipulate events: they may be *resolved* into an action on some storage, *forwarded* to another representative, *deferred* in a buffer for later handling, *changed* or *combined* or *discarded*. Taken together with event descriptions these processes form the building blocks of a simple specification framework for SADT representations which is sufficiently formal to allow automatic skeleton code generation and a certain amount of rigorous analysis to be applied to each representation[8]. This formal basis is essential for reasoning in the presence of weak coherence.


Repository object (monitor)
Network
Read
Update
Unitary - all operations centralised
Partitioned - data split between representatives
Broadcast
Replicated - local reads and broadcast updates
Weakened - read caches and update buffering
Figure 1. Different representation schemes for an SADT

3.3 Programmability

The creation of an SADT is obviously more complex than ordinary object creation, since a distributed family of objects must be created. We encapsulate this complexity by using an *abstract factory* class for each SADT[11]. The factory combines a language for expressing constraints – such as the degree of weakness acceptable to an algorithm or a pattern of use in the operations – with a decision procedure to determine which of the available representations of the SADT is appropriate to best satisfy these constraints. This decouples the algorithmic constraints from the strategy used to select a representation, allowing both constraint language and representation populations to evolve over time.

Sub-typing is an essential component of large-scale object-oriented applications. Sub-types may derived from SADT front-end objects in the usual way, with the SADT environment creating the appropriate factory object. Sub-types have access to all the representations available to the parent type, and possibly to extra, more specific representations. New representations for SADTs are generated by providing event-handling code and extending the factory decision procedure to allow the new representation to be selected when appropriate constraints are met. Event handlers have access to the core events, which are implemented by the common super-type of representatives. Completely new SADTs may be defined similarly, starting from a description of the type-specific events.

The prototype environment makes extensive use of Modula-3's generic interfaces and powerful configuration control language, together with simple generation and analysis tools.

3.4 Example: the Accumulator SADT

The accumulator[12] is a type which combines a single value with an update function. The value is updated by combining the submitted new value with the current value through the update function. An example is a minimising accumulator, where the update function returns the smaller of the current and submitted values.

The simplest accumulator stores the value in a single representative, performs all updates as they are received, and always returns the current value. Variations[9] allow the semantics to be weakened with various combinations of replication, update deferral and value combination. These variations lead to subtly different accumulators, which may be more efficient than the simple case whilst still providing acceptable guarantees for applications.

4 TSP with SADTs

To illustrate the issues outlined above we present a case study of implementing Little's algorithm on a network of workstations. The starting point for the application code is a sequential version of the program, which uses two key abstract data types:

1. a *priority queue* used to hold the population of nodes. Nodes are dequeued from to be expanded, and the resulting child nodes enqueued. The priority property keeps the "best" node as the head element; and
2. an *accumulator* used to hold the length of the shortest known tour.

The user algorithm functions exactly as outlined in section 2.

4.1 Direct Implementation

A parallel application can quickly be produced by converting the priority queue and accumulator into SADTs and using a number of identical worker processes co-ordinated through these two abstractions.

The simplest implementation of the required SADTs is to implement an object in one address space in the system, and use the mechanisms of the runtime system to provide access to this object across the system. By protecting the object with a monitor, sequential consistency for the SADT is maintained. We term this the *unitary* or *shared object* style.

A problem immediately arises with this simple implementation. Termination in the sequential case is signalled by the priority queue being empty and unable to service a *dequeue* request; in the parallel case this condition may arise in the normal course of the algorithm and correctly detecting termination involves predicting when the structure *is and will remain* empty. This can be achieved by the use of an additional accumulator counting the number of paths expanded.

The application was run on a set of SGI O2 workstations connected by a 10MBit/sec Ethernet running TCP/IP – a platform providing high compute performance but relatively poor communication. Achieving reasonable performance for a fine-grained algorithm such as the TSP solver therefore poses a significant challenge. Indeed, the simple implementation actually exhibits a considerable slow-down as processors are added! Profiling the application exposes the reasons for this. Figure **2** shows a percentage breakdown of the execution time of the application on a 4-processor system by the different SADT operations that are invoked.

`Shortest` is the accumulator for the current best-known solution, `Counter` is the accumulator used for termination detection, and `Store` is the priority queue. The final columns on the histogram represents the available CPU time remaining to devote to the application problem once the SADT operations have been taken into account. Clearly, the operations on the SADTs dominate the runtime of the application, leaving a very small proportion of the time for running the actual application-specific code.

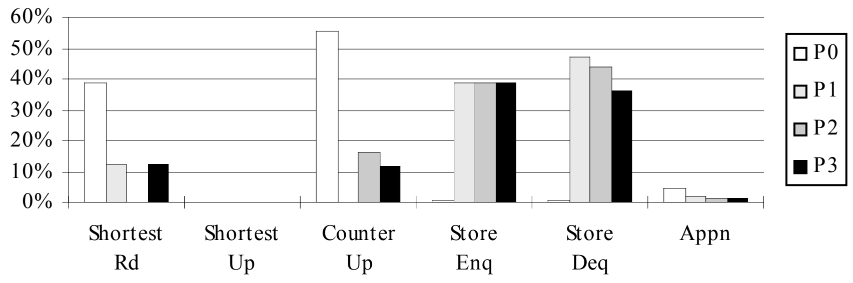


Figure 2. Profiling of naïve implementation.

4.2 Optimised Implementation

To improve the performance of the parallel application, the excessive costs of the SADT operations must be addressed. We may make a number of observations about the TSP algorithm. Firstly, there are significant patterns in the use of the shared data. We may reduce overheads by matching an appropriate implementation of the SADTs to their observed patterns of use. Secondly, several of the "normal" coherence constraints are not actually required for the algorithm to function correctly, and relaxing them again gives scope for performance improvements. Relaxation trades-off a decrease in the efficiency of the heuristics against reduced overheads in maintaining the shared data abstraction. We implement these optimisations by making additional, optimised representations of the SADTs, and the appropriate decision procedure, available to the run-time environment.

4.2.1 Replicated and Weakened Accumulators

From inspection it may be seen that the accumulator holding the shortest known tour is read each time an element is read from the priority queue, to see whether the path should be pruned; it is only updated when a shorter complete tour has been found, which typically happens only a few times during the application. The frequency of reads is much higher than that of updates, so by replicating the accumulator (figure 1) a read may be rapidly serviced locally.

By broadcasting updates the accumulator remain strongly coherent. Weaker approaches apply updates locally and periodically synchronise the accumulator to propagate the best global value, or apply several updates *en bloc* in a single network transaction.

4.2.2 Partitioned Priority Queue

Similar arguments apply to the priority queue: it may be distributed across the system, and relax the requirement that a dequeue operation returns the overall best element to allow the element returned to be locally best but not necessarily globally optimal. Distribution also makes the memory of all nodes available for representing the queue, rather than bounding its size to the memory of a single node. A background task at each representative periodically shuffling elements randomly between representatives. This shuffling improves data balance and, by shuffling high-priority elements, ensures that the algorithm as a whole focuses on the elements with the highest global priority. An important optimisation of shuffling is not to shuffle the *highest* priority elements, but instead to shuffle from one or two elements down. This ensures that a high-priority element is expanded, and is not constantly shuffled.

4.2.3 Synchronising Priority Queue

Termination relies on the counter accumulator, which must be strongly coherent in order to solve the consensus problem[13]. In the single-bus system the unitary implementation appears to be the best that can be done: once the other SADTs have been optimised, however, operations on the counter dominate the performance of the code.

The original reason for introducing the counter accumulator was to solve the distributed termination detection in the face of transient emptiness in the queue. A key observation is that termination can be correctly inferred when the queue *is and will remain* empty, so no new elements can appear "spontaneously" due to other threads. Without additional information about the algorithm, the only condition of the SADT in which we know this to be the case is when *all* processes sharing the queue are attempting to dequeue from it: this is the termination state, as all work has been consumed and no more can arise. On detecting this condition we can release all the processes simultaneously – by returning `NIL`, a logical extension of the semantics of the sequential dequeue operation to the concurrent case.

The interface to the priority queue is extended so that each thread sharing the SADT is known. A mechanism is then implemented to detect when a representative is empty and all local threads are performing dequeue operations. Combining these local results using a consensus protocol completes the system and removes the need for the counter accumulator.

This insight – the usefulness of *scoped deadlocks* – appears to capture a very general property of termination in parallel algorithms which we are investigating further.

4.2.4 Performance

When more optimal SADTs are used, performance improves dramatically (figure **3**). It is clear that the optimisation of the SADTs results in a dramatic increase in the proportion of the total CPU time available for running the (unchanged) application code.

Figure **4** shows the speedup of the application relative to 1 processor when running on 2, 4, 8 and 12 O2 workstations solving a 21-city problem. Real speedups are now achieved. The curve *Optimised SADTs 1* shows the performance obtained when using a random data balancing strategy within the priority queue SADT; *Optimised SADTs 2* shows the performance obtained using simple load information to reduce the chances of starvation..

The *Indy SADTs 2* curve shows the speedups achieved when running the same code on a network of SGI Indy workstations. The Indy has roughly half the raw compute speed of the O2 with the same communication bandwidth, giving a better computation-to-communication balance. (This curve is normalised to a speedup of 2 on 2 processors, as the single processor code exceeds physical memory limits and incurs overheads that are avoided using several processors.)

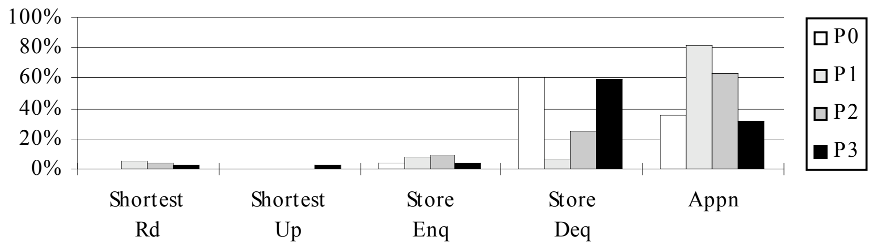


Figure 3. Profiling of the fully optimised implementation.

It is obvious that the improvements in performance trail off with increasing system size. The reason for this is that the priority queue's balancing strategy becomes less effective as size increases, due to both the inadequacy of algorithm and to the limited performance of the underlying network transport. The shuffling on which the current strategy is based results in a network load of about 80KBytes/sec per workstation: a small number of workstations can thus saturate the Ethernet. This situation could be improved by a more informed shuffling strategy, a higher bandwidth transport, or a transport with better scaling properties such as a point-to-point ATM network – all issues demanding further work.

The code implementing the algorithm is almost identical in sequential and parallel systems – the only extra code being algorithmic pattern and coherence constraints. The resulting parallel application achieve high performance for a fine-grained algorithm on an coarse-grained architecture. There is obviously significant scope for improving performance further: however the key principle of optimising performance through constrained representation selection for otherwise fully abstract SADTs has been clearly demonstrated.

5 Related Work

A great many systems have been proposed to allow objects to be manipulated concurrently and location-independently – notable are the CORBA standard, the Orca language[1] and Modula-3 network objects. These systems provide the mechanisms required to build shared data structures, but do not directly address the design of efficient systems. The "single abstraction *via* a community of objects" approach builds on this substrate: it is used by SADTs as well as a number of other systems[2][4][7][10], and is perhaps best exemplified by Concurrent Aggregates[5].

A slightly different approach is taken by languages such as HPF and HPC++, providing a library of effectively sequential types together with a small number of *bulk operations*. The operations hide their efficient parallel implementations from the programmer (both syntactically and semantically), preventing exploitation of more relaxed coherence models.

Berkeley's Multipol[21] provides a library of types for irregular computations. A Multipol object is *multi-ported* in the sense that it exports both a global view and a view of the locally-held data. The SADT approach hides the local view, substituting bulk operations and weakening instead.

Figure 4. Performance of optimised scheme

The canonical example of weakening is the Bulk Synchronous Parallelism model, in which processes' view of shared state may drift between global synchonisation points. Recent work[19] has defined a set of BSP collection types, which provide bulk operations and high-level interfaces but are still constrained to use the globally-agreed coherence model.

6 Conclusion

We have presented a programming environment based around a collection of types providing a sound basis for parallel programming, whose characteristics are well-matched to the network-of-workstations architecture. Common programming techniques are incorporated in a semantically consistent way, allowing them to be deployed without unduly complicating the application code. Preliminary results indicate that the system can achieve relatively high performance by progressive refinement encapsulated within the run-time system. We are currently performing more extensive evaluations based on the "Cowichan problems" – and it is encouraging to note that the problems identified in these applications are well addressed by the SADT approach[20].

A major future direction is to improve our ability to detect patterns of use of an SADT for which a known efficient representation exists. This would free the programmer of the need to provide hints to the run-time system, at the cost of typing the library closely to the analysis system. We are also seeking means of integrating more closely into the type system the semantic constraints currently supplied to SADT factory objects.

7 References

[1] Henri Bal, Andrew S. Tanenbaum and M. Frans Kaashoek, "*Orca: a language for distributed programming,*" ACM SIGPLAN Notices **25**(5) (May 1990) pp.17-24.

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

[3] Andrew Birrell, Greg Nelson, Susan Owicki and Edward Wobber, "*Network objects,*" Research report 115, Digital SRC (1994).

[4] John Chandy, Steven Parkes and Prithviraj Banerjee, "*Distributed object oriented data structures and algorithms for VLSI CAD,*", pp.147-158 in Parallel algorithms for irregularly structured problems, **LNCS 1117**, ed. A. Ferreira, J. Rolim, Y. Saad and T. Yang, Springer Verlag (1996). Proceedings of IRREGULAR'96.

[5] A.A. Chien and W.J. Dally, "*Concurrent Aggregates,*" ACM SIGPLAN Notices **25**(3) (March 1990) pp.187-196.

[6] John Davy, Peter Dew, Don Goodeve and Jonathan Nash, "*Concurrent sharing through abstract data types: a case study,*", pp.91-104 in Abstract machine models for parallel and distributed computing, ed. M. Kara, J. Davy, D. Goodeve and J. Nash, IOS Press (1996).

[7] Simon Dobson and Andy Wellings, "*A system for building scalable parallel applications,*", pp.218-230 in Programming environments for parallel computing, ed. N. Topham, R. Ibbett and T. Bemmerl, North Holland Elsevier (1992).

[8] Simon Dobson and Chris Wadsworth, "*Towards a theory of shared data in distributed systems,*", pp.170-182 in Software engineering for parallel and distributed systems, ed. I. Jelly, I. Gorton and P. Croll, Chapman and Hall (1996).

[9] Simon Dobson, "*Characterising the accumulator SADT,*" TallShiP/R/22, CLRC Rutherford Appleton Laboratory (1996).

[10] Stephen Fink, Scott Baden and Scott Kohn, "*Flexible communication mechanisms for dynamic structured applications,*", pp.203-213 in Parallel algorithms for irregularly structured problems, **LNCS 1117**, ed. A. Ferreira, J. Rolim, Y. Saad and T. Yang, Springer Verlag (1996). Proceedings of IRREGULAR'96.

[11] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "*Design patterns: elements of reusable object-oriented software,*" Addison-Wesley (1995).

[12] Don Goodeve, John Davy and Chris Wadsworth, "*Shared accumulators,*" in Proceedings of the World Transputer Congress (1995).

[13] Maurice Herlihy, "*Wait-free synchronisation*," ACM Transactions on Programming Languages and Systems **11**(1) (1991) pp.124-149.

[14] John D.C. Little, Katta G. Murty, Dura W. Sweeney and Caroline Karel, "*An algorithm for the travelling salesman problem*," Operations Research **11** (1993) pp.972-989.

[15] W.F. McColl, "*Bulk synchronous parallel computing,*" pp.41-63 in Abstract machine models for highly parallel computers, ed. J.R. Davy and P.M. Dew, Oxford Science Publishers (1993).

[16] Greg Nelson, "*Systems programming with Modula-3,*" Prentice Hall (1993).

[17] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling, "*Numerical recipes in C*," Cambridge University Press (1989).

[18] V.J. Rayward-Smith, S.A. Rush and G.P. McKeown, "*Efficiency considerations in the implementation of parallel branch-and-bound*," Annals of Operations Research **43** (1993).

[19] K. Ronald Sujithan and Jonathan Hill, "*Collection types for database programming in the BSP model,*" in Proceedings of IEEE Euromicro (1997).

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

[21] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Jeff Jones, Arvind Krishnamurthy and Chih-Po Wen, "*Parallel data structured for symbolic computation,*" in Workshop on Parallel Symbolic Languages and Systems (1995).