Distributed Shared Abstractions (DSA) on Multiprocessors *

Christian Clémençon (clemenco@lse.epfl.ch)
Bodhisattwa Mukherjee (bodhi@watson.ibm.com)
Karsten Schwan (schwan@cc.gatech.edu)

GIT–CC–93/25

Abstract

Any parallel program has abstractions that are shared by the program's multiple processes, including data structures containing shared data, code implementing operations like global sums or minima, type instances used for process synchronization or communication, etc. Such shared abstractions can considerably affect the performance of parallel programs, on both distributed and shared memory multiprocessors. As a result, their implementation must be efficient, and such efficiency should be achieved without unduly compromising program portability and maintainability. Unfortunately, efficiency and portability can be at cross-purposes, since high performance typically requires changes in the representation of shared abstractions across different parallel machines.

The primary contribution of the DSA library presented and evaluated in this paper is its representation of shared abstractions as objects that may be internally distributed across different nodes of a parallel machine. Such *distributed shared abstractions (DSA)* are encapsulated so that their implementations are easily changed while maintaining program portability across parallel architectures ranging from small-scale multiprocessors, to medium-scale shared and distributed memory machines, and potentially, to networks of computer workstations. The principal results presented in this paper are (1) a demonstration that the fragmentation of object state across different nodes of a multiprocessor machine can significantly improve program performance and (2) that such object fragmentation can be achieved without compromising portability by changing object interfaces. These results are demonstrated using implementations of the DSA library on several medium-scale multiprocessors, including the BBN Butterfly, Kendall Square Research, and SGI shared memory multiprocessors. The DSA library's evaluation uses synthetic workloads and a parallel implementation of a branch-and-bound algorithm for solving the Traveling Salesperson Problem (TSP).

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

*This work was supported in part by the Swiss National Science Foundation and by NSF grant CCR-8619886.

1 Introduction

A parallel program can be viewed as a set of independent processes interacting via shared abstractions. Such abstractions include shared data, shared types like work queues and locks, as well as globally executed operations like global sums, merging scan-lines into coherent bitmaps, and others. Since the shared abstractions used in a parallel program represent the program's global information, their efficient implementation can be crucial to the program's performance, its scalability to different size machines, and its portability to different target architectures. For example, a port of a parallel program from a shared memory to a distributed memory machine typically requires the re-implementation of its shared abstractions from using explicit synchronization constructs to sending messages across statically or dynamically defined communication structures linking the program's processes[41, 15]. Similarly, the differences in local to remote memory access costs in most larger-scale parallel machines (i.e., the NUMA properties of such machines[28]) require substantive changes in the implementation of synchronization constructs for small-scale parallel machines like Sequents[2] or Silicon Graphics multiprocessors to larger-scale parallel machines[31].

The contribution of our work toward increased scalability and portability of parallel programs is the provision of the DSA library for the efficient implementation of objects termed "Distributed Shared Abstractions (DSA)". Scalability on SMP machines is achieved by implementation of such objects as sets of object fragments[41, 45] linked by a user-defined communication structure, which we term a *topology*. The resulting parallel program's portability is improved by encapsulation of its shared abstractions as objects with operational interfaces that may remain invariant across objects' different implementations for specific target machines.

Distributed shared abstractions (DSA) permit programmers to:

- define and create encapsulated objects that may be internally fragmented in order to take advantage of localities of reference to locally vs. remotely stored object state, and
- employ library mechanisms for implementing efficient, abstraction-specific communications among object fragments, thereby exploiting their application-level knowledge about the semantics of object invocations and the communication patterns among object fragments.

DSA support is implemented as a runtime library layered on top of a Mach-compatible Cthreads package developed by our group[7, 33]. As a result, a parallel program written with the DSA library consists of a set of independent threads interacting via DSA objects. Implementation of a DSA object itself involves (1) the storage of object fragments (state and code) at participating processors and (2) communication between these processors and fragments via a remote invocation mechanism, also part of the library. Portability of the DSA library to different shared memory multiprocessors is due to the portability of the underlying Cthreads library. Portability of DSA-based programs from shared to distributed memory machines (including workstation networks) is due to the use of an easily ported remote invocation mechanism[5] for communication among object fragments.

The benefits derived from constructing DSA objects are manifold, including (1) potential reductions in contention of access to an object, since many operations on the object will access only locally stored copies of its distributed state, (2) decreases in invocation latencies, since local accesses are faster than remote accesses, and (3) the ability to implement objects such that

they may be used on both distributed and shared memory platforms, therefore increasing the portability of applications using them. Performance benefits derived from the fragmentation of objects or object state have been shown possible for many implementations of higher level operating system services in distributed systems (e.g., file systems[38]) and for application-specific services on distributed memory machines[41]. For shared memory multiprocessors, similar results have been attained for RPC implementations on NUMA machines like the BBN Butterfly multiprocessor[28] and are demonstrated in this paper for a program-specific abstraction (i.e., a shared queue) in a parallel branch-and-bound application executed on a 32-node GP1000 BBN Butterfly and a 32-node Kendall Square Research KSR-1 supercomputer. In this shared queue, alternative fragmentations of the object make use of application-level information about both the specific pattern and the rates of communications between multiple queue fragments. Sample application-level knowledge includes desirable or acceptable global or local orderings among queue elements, tolerable delays regarding the propagation of information among queue fragments, etc.

There are several differences of our research to current work on distributed shared abstractions. First, in contrast to recent research in cache architectures for parallel machines (e.g., the DASH project[16]) and in weakly consistent distributed shared memory[4, 22, 29], we do not assume a fixed model (or limited number of models) of consistency between object fragments. Instead, programmers can implement object-specific protocols for state consistency among object fragments, using the low-level remote invocation mechanism offered by the DSA library. Second, since communications among objects fragments are explicitly programmed, shared abstractions implemented with the library are not subject to some of the performance penalties in distributed shared memory systems arising from sharing multiple, small abstractions allocated on a single shared page (i.e., false sharing leading to additional and/or excessively large communications). Conversely, by adding calls like "invalidate(page)" and "get(page)"[22], etc. to our current low-level communication calls, distributed shared memory (DSM) abstractions may be implemented and compared with alternative representations within the existing DSA library. Such an implementation and performance results attained on a cluster of workstations are described in [27]. Third, ongoing research on distributed objects is contributing language and compiler support for describing objects and then compiling object interactions into efficient runtime invocations, using custom communication protocols[3] and/or exploiting active message paradigms[25, 48, 21]. We share with such work the assumption that communications between different object fragments can often benefit from the use of active messages and that the use of active messages can improve the locality of parallel programs[47]. However, we also posit that the compilation techniques and runtime support described in such work should be enhanced to support object fragmentation as well as other well-known techniques for optimizing object performance (e.g., caching, the use of knowledge about object semantics expressed by attributes[36], etc.). Fourth, in contrast to the research of Shapiro on fragmented objects[46], we explicitly consider the communication structure linking object fragments in order to exploit application-specific knowledge of the object's communication patterns.

The fifth difference of our work to other research concerns our previous kernel-level implementation of DSA functionality on hypercube machines[41]. In contrast to those implementations, the layering of DSA objects on a basic remote invocation mechanism has resulted in library portability to various target platforms, including the aforementioned shared memory platforms and a recently completed implementation on a network platform[27]. Last, shared abstractions are easily instrumented, evaluated[41, 37, 26], and even dynamically adjusted, us-

ing custom[35] or library-provided[19] mechanisms for on-line program monitoring and without exposing such instrumentation to application programs[35].

The remainder of this paper first presents a sample parallel application and the abstractions shared by its concurrent processes (Section 2). Next, the performance effects of alternative, shared memory implementations of such shared abstractions are evaluated experimentally on 32-node BBN Butterfly and KSR multiprocessors. In Section 3, the same abstractions are implemented using the DSA library developed in our research. In Section 4, the DSA library is described and evaluated in detail. Section 5 compares our work with related research and finally, Section 6 describes our conclusions and future research.

2 Shared Abstractions in Parallel Programs

Programmers use a variety of methods for decomposition of programs into concurrently executable processes, including the static or dynamic decomposition of programs' data domains, divide and conquer strategies, functional decompositions, and pipelining. Many parallel programs resulting from such decompositions exhibit coordinator/server structures, where coordinator processes generate work units processed by workers[24] or at least supervise a number of workers. Sample applications structured in this fashion range from (1) domain-decomposed scientific applications to (2) MultiLisp implementations on parallel machines, where "futures" are entered into queues and removed and processed by available processors[20], to (3) parallel optimization codes[13, 40], and (4) even operating system services like file or I/O servers.

The sample parallel program used in our research is a client/server structured application, a parallel branch-and-bound algorithm solving the Traveling Salesperson problem (TSP). We employ the algorithm of Little, Murty, Sweeney and Karel (LMSK algorithm)[23], and we use a parallelization first described in [32]. The resulting parallel algorithm essentially conducts a search in a dynamically constructed search space, where two abstractions are shared among searcher threads: (1) a global best tour value, which is used for pruning the search space, and (2) a work sharing abstraction for the dynamic distribution of work among the searcher threads.

The sample parallel algorithm described in this paper is interesting for three reasons. First, branch-and-bound algorithms are commonly used in the solution of optimization problems and have therefore, been frequently studied and evaluated on parallel machines. Second, experimental evaluations of the algorithm's implementation on distributed memory platforms[43, 40] and on workstation networks[13] have already demonstrated the importance of the work sharing and tour abstractions to parallel program performance, where different implementations of the queue itself and of load balancing among queue fragments significantly effect speedup and scalability. In part, this importance is derived from the relatively fine granularity of this application, where few computations are necessary between communications. Not all branch and bound applications have this property. Third, this paper demonstrates that the efficient implementation of both shared abstractions is equally important on NUMA multiprocessors, including the BBN Butterfly machine and the KSR supercomputer.

2.1 The LMSK Algorithm

The traveling salesperson problem is to find the least cost round trip tour of a salesperson who must visit each of N cities. There is an integer cost $c(i,j)$ to travel from city i to city j, where

total cost is the sum of the individual costs along the edges of the tour. The problem to be solved is represented as a $N \times N$ cost matrix encoding the directed graph being traversed.

The LMSK algorithm partitions the original problem into progressively smaller subproblems, which are represented as nodes of a dynamically constructed search tree. The algorithm computes a lower bound on the cost of the best tour in each node, and then expands the search tree incrementally toward the goal node, using two heuristics to guide the search. Specifically, starting with the root node representing the original problem, the algorithm repeatedly executes the following steps: (1) it selects a node from among all leaf nodes of the current tree (node selection heuristic), (2) it chooses an edge (i,j) from the cost matrix associated with the selected node (edge selection heuristic), (3) it expands the selected node into two child nodes, the right child including the selected edge among its possible tours, the left child excluding it, and (4) it computes for each child node the lower bound cost of all possible tours defined by the child. The size of the right child (the number of cities to be visited) is decreased by one compared to its parent node. The algorithm continues to choose leaf nodes and expand the search tree until a tour has been found (i.e., a leaf node is of size 2). Once a tour is found, the tree representing the search space may be pruned by deletion of all leaf nodes with lower bounds greater than or equal to the value of the found tour. When all leaf nodes have been expanded or pruned, the lowest of all the found tours is the solution of the problem. A more complete description of the algorithm can be found in [23].

2.2 Parallel Implementation of the LMSK Algorithm

Our parallel LMSK algorithm is implemented as a collection of asynchronous, cooperating searcher threads each of which independently executes the algorithm's main procedure. The resulting code is outlined in Figure 1 (it implements the LMSK search algorithm as described in Section 2.1). A searcher executes the program steps during each iteration of the while loop. It repeats this process until it reaches consensus with all other searchers that the best tour has been found. The searchers cooperate using two shared abstractions: (1) a work sharing queue ("work_queue") storing the leaf nodes of the tree representing the search space and permitting the dynamic distribution of work among searchers, and (2) a shared integer value ("best_tour") representing the current best tour found so far and used by searchers to prune the search space.

A TSP computation is initiated by a single thread, by first enqueuing a representation of the initial problem (the root node) in the work sharing queue, and then forking some predefined number of searcher threads. The computation terminates upon completion of all searcher threads.

2.3 Shared Memory Implementation of TSP Abstractions

When using shared memory to implement the TSP "tour" and "work queue" abstractions, two important factors affect the resulting parallel program's performance: (1) contention due to concurrent abstraction access (synchronization overhead) and (2) remote memory access costs (communication overhead). For example, for the BBN Butterfly, the ratio of access costs to local vs. remote memory is approximately 1:12, which implies that the costs of executing an operation on a shared abstraction strongly depends on the number of remote references performed by the operation[6]. Since this ratio tends to be even worse on modern NUMA machines and in order to reduce contention and take advantage of locality, implementations of shared

```
void find_best_tour( work_queue, best_tour )
tsp_queue_t work_queue;
tsp_tour_t best_tour;
{
        tsp_node_t node, left_node;
        int i, j;

        while( node = work_queue->get( work_queue ) ) {
                if( node->size == 2 )
                        best_tour->new( best_tour, node->lower_bound );
                else if( node->lower_bound < best_tour->read( best_tour ) ) {
                        matrix = rebuild_matrix( node );
                        choose_best_edge( matrix, &i, &j );
                        expand_left( node, left_node, i, j );
                        work_queue->put( work_queue, left_node );
                        expand_right( node, matrix, i, j );
                        work_queue->put( work_queue, node );
                }
        }
}
```

Figure 1: The Parallel LMSK Algorithm

abstractions in NUMA machines often explicitly distribute their state and code data to participating processors' memory units, and then use abstraction-specific communication structures to maintain consistency among such distributed information. Alternative implementations of the shared work queue abstraction are described and evaluated next. We continue to use shared memory for implementation of the "tour" abstraction, since the performance impacts of alternative implementations of this abstraction are small in our application (this may not hold when tours are found more frequently and/or for implementations on distributed memory machines, as described in [41]).

The work sharing queue implementation for TSP stores the current leaf nodes of the search tree. In a parallel implementation, this abstraction implements: (1) a node selection heuristic, (2) a work distribution strategy, and (3) a protocol for program termination. The node selection heuristic is implemented as an ordering of queue elements. Queue elements (nodes) are ordered (a) by their lower bound on the problem's solution and (b) by their sub-problem sizes. When using a double-priority queue ordered by (a) and (b), retrieval and processing of the first node on the queue implements a best first heuristic for node selection. In other words, best first node selection always chooses for expansion the node with the least subproblem size from the set of nodes that have the least lower bound value. This strategy favors nodes that are likely to lead to good solutions fast. It has been shown useful in other LMSK implementations[40], since it tends to minimize the total number of nodes present in the final search tree.

We term a queue implementation *consistent* if a global priority ordering is maintained among queue elements. A consistent queue faithfully implements the 'best first' node selection heuristic, whereas queue implementations that do not maintain a total queue ordering – termed

inconsistent – decrease the effectiveness of the node selection heuristic. Decreased effectiveness is undesirable since it leads to substantial additional computations in the parallel algorithm due to the expansion of nodes that would not be expanded by the sequential algorithm – termed *additional nodes*.

Since the TSP's search space is constructed dynamically, another important role of the work sharing abstraction is to ensure the equal distribution of work (i.e., nodes) among searchers. This is trivially ensured when using a global queue. In fragmented queue implementations, however, load-balancing must be performed among queue fragments. Since such load balancing must take into account both the sizes of queue fragments (number of nodes per fragment) and the ordering among nodes, it is henceforth termed *quality balancing*. An effective quality balancing strategy, then, ensures both a global ordering of nodes and an equal distribution of nodes among queue fragments. Tradeoffs in effectiveness vs. efficiency of queue implementation and quality balancing are apparent in three alternative queue implementations on the BBN Butterfly; they will be evaluated experimentally in Section 2.4.

a) 'Global' queue representation: A first implementation using a single queue copy takes advantage of shared memory. Each searcher thread allocates new nodes in its processor's local memory. However, all such nodes are linked into a single queue that spans all processors' memories. A predetermined, single processor maintains the queue's head as well as a spin lock for mutual exclusion in queue access. In this implementation, no work distribution strategy is needed, and the termination protocol is implicit: searchers terminate when the queue is empty.

b) Distributed representation without quality-balancing: A second implementation attempts to maximize locality of access to queue elements, while performing minimal load balancing. Specifically, the global priority queue is split into several subqueues, which are interconnected via a unidirectional ring. Each searcher thread owns a local queue fragment, which is implemented as a priority queue and protected by a local spin lock. The searcher thread enters and removes nodes into/from its local subqueue, and allocates new nodes in local memory. The work distribution strategy performs load-balancing as follows: if a searcher performs a 'get' operation on an empty local queue fragment, it then simply removes the 'best' node from the next non-empty remote queue fragment along the ring. This results in the sharing of 'good' nodes among searchers only when searchers have exhausted their own parts of the search space. This queue representation also requires an explicit termination protocol. In this case, a searcher terminates when all of the queue fragments along the ring are empty and at least one tour has been found.

c) Distributed representation with quality-balancing: A third implementation is like the previous one, but also performs continuous quality-balancing. Specifically, similar to the strategy used by Felten in [12], every two 'get' operations by a searcher thread on its local queue trigger a move of the second best node from the local queue to the next subqueue along the ring. As a result, 'good' nodes are frequently shared among different searcher threads. This increases the overall quality of nodes used by searcher threads, but it also increases the total number of accesses made by threads to non-local node representations[1].

2.4 The Scalability of Parallel Programs: A Case Study of Shared Queues

All measurements given in this section are performed on a 32-node GP 1000 BBN Butterfly. The measurements shown are the averages of the executions of 100 different, randomly generated

[1] Sharing of nodes more (for every 'get' operation) or less (every four 'get' operations) frequently results in performance degradation. Similarly, the association of node sharing with 'getting' vs. 'putting' nodes appears to have no visible performance effects. Use of a doubly linked list and the resulting sharing of nodes with two neighbors may increase performance in larger-scale shared memory multiprocessors, but will (1) decrease performance in the DSA library implementation of TSP due to the required additional remote invocations and (2) may decrease performance in message based systems due to the required additional message operations[41].

TSP problems. Each TSP problem has 32 cities and is described by an initial random cost matrix, with costs in the range of 1 to 50. Each TSP problem is executed for each of the three work sharing abstractions, with the same initial cost matrix. Each searcher thread executes on its own dedicated processor with local copies of its code, stack, local data, and with a local copy of the cost matrix.

The first set of experimental results shown below demonstrate that the achievement of good *scalability* of parallel programs must use representations of shared abstractions that take into account a program's semantics as well as its implementation details. Specifically, in Figures 2 and 3, we show the execution times and speedups of the TSP application when it is executed with 1 to 25 processors using each of the three different queue implementations. *Variant "global"* is the global queue implementation, where searcher threads share all subproblems ranked by knowledge about program semantics, which is subproblem size and quality. In contrast, *variant "distributed"* is the distributed queue without quality balancing, where searcher threads share no knowledge concerning such program semantics. *Variant "distributedQB"* is the distributed queue with quality balancing. Speedup is computed as the ratio between sequential and parallel execution times, where sequential execution cost is determined by execution of the application on a single processor with a single thread that does not experience any of the synchronization costs arising for multiprocessor executions.

The results depicted in Figure 3 demonstrate that significant execution speedups are possible with the distributed queue implementations (execution time of the sequential implementation of the application is 18484 milliseconds). Similar speedups should be achievable on larger parallel machines as long as the problem size is increased beyond the 32 cities used in our measurements.

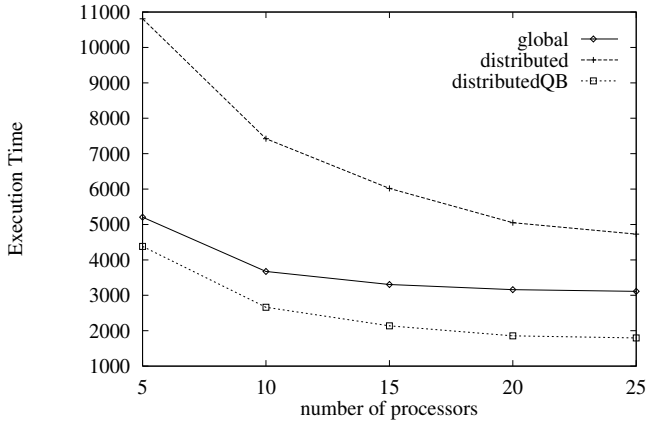

Figure 2: Execution times (milliseconds) of variants of the TSP application

It is apparent from both Figures 2 and 3 that *variant distributedQB* – the distributed queue with quality balancing – behaves best. In other words, while improvements in locality of access to queue elements exist in *variant distributed* compared to *variant global*, the disadvantages incurred by additional work performed by searcher threads outweigh the accrued performance gains. In effect, unless tours are found, in *variant distributed* each searcher thread ignores the information about the search space available to other searchers. Therefore, while the distrib-

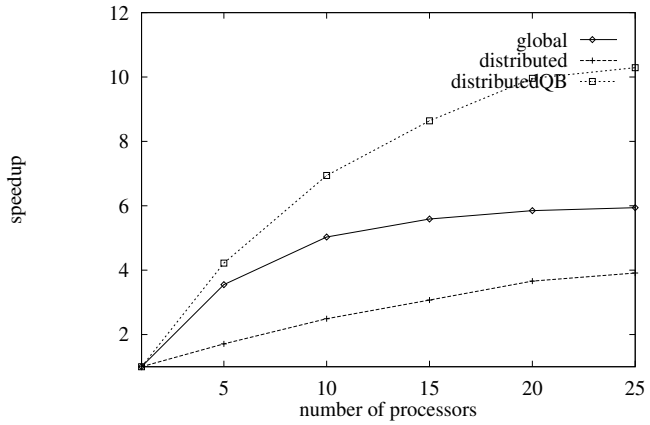

Figure 3: Speedups of variants of the TSP application

uted implementations (*variants distributed and distributedQB*) are superior to *variant global* regarding the locality of access, the complete loss of the total ordering maintained by the global queue in *variant global* is not acceptable. In other words, it is not an effective strategy to implement shared abstraction without using information about program semantics. This may be stated as the first important insight from these results:

- To attain acceptable performance, it may be critical to use information about program semantics in the implementation of distributed shared abstractions.

These results and similar results reported for distributed memory machines[13, 40] are our main motivation for rejecting conceptually simpler approaches like distributed shared memory[4] for the implementation of shared abstractions in parallel programs.

Figure 4 provides additional explanation of the results depicted in Figures 2 and 3, by depicting the total number of nodes expanded in order to arrive at a solution. As stated in the previous paragraph, the total number of expanded nodes is highest when load sharing ignores semantic information in *variant distributed* (i.e., no quality balancing), whereas the number of expanded nodes with quality balancing (*variant distributedQB*) closely approximates the number attained with the globally ordered priority queue (*variant global*).

The importance of alternative queue implementations to parallel TSP performance is further underlined by the measurements depicted in Figure 5. They demonstrate the second insight we derive from the measurements presented in this section:

- The development of scalable parallel programs requires ease of change in the implementation of abstractions shared by multiple application processes.

The demonstration of this fact presented here concerns the effects of alternative queue implementation. Specifically, we measure the ratio of the time searchers spend in the queue abstraction vs. their total execution times. The elapsed time in the queue abstraction consists of time spent in the 'get' and 'put' operations, which can be decomposed into: (1) the time spent for managing the queue, (2) the time spent for explicit communication between queue

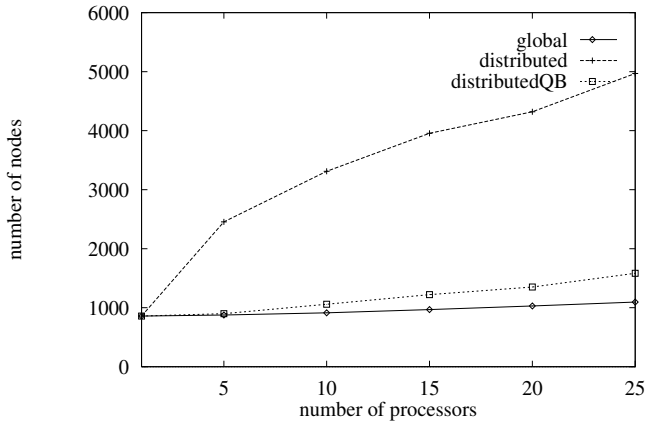

Figure 4: Total number of node expansions

fragments (generating messages, processing requests, etc.), (3) the time spent waiting for locks protecting the queue from concurrent accesses, and (4) the wait time experienced during termination detection. Of these times, (1) is insignificant since the time spent managing the double priority queue is only about 1.75% of total program execution time. In *variant global*, the time spent in the work sharing abstraction is almost entirely due to (3) – queue access contention. This time significantly increases with the number of processors and beyond 15 processors, it exceeds the time spent doing useful work (i.e., expanding nodes). It is the main cause for the degradation of speedup in *variant global* as shown in Figure 3.

In *variant distributed*, contention is insignificant, because searchers almost always access local queue fragments and therefore, the time searchers spend in the queue is primarily due to queue management and termination detection, neither of which are very time-consuming. As expected, the quality balancing performed in *variant distributedQB* increases the time spent in the queue abstraction, but it is outweighed by the significant reduction in the total number of node expansions performed during problem solution.

An issue not discussed above is the storage of node data, which results in performance differences regarding the expansion of locally vs. remotely stored nodes. In this implementation of TSP on the BBN Butterfly machine, such differences are not as significant as in distributed memory implementations[13, 40]. Expansion of a locally vs. remotely stored node can be performed in 25 milliseconds vs. 27 milliseconds.

To summarize, we have used the shared queue abstraction in a parallel branch-and-bound program to demonstrate that the TSP program's speedup is limited by the performance of the abstractions shared by its processes. This demonstration is important since it provides evidence that suitable performance cannot be attained without permitting programmers to employ application-level knowledge about abstraction usage when implementing shared abstractions. One way of using such knowledge is the development of compile-time or even runtime configurable abstractions. This paper shows that even complex implementations of such configurable abstractions can significantly improve program performance. In this section, for example, we show that the most complex implementation of the major shared abstraction in TSP – the

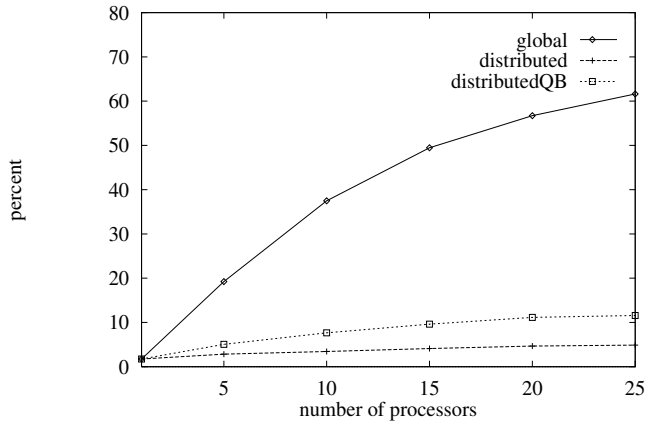

Figure 5: Percentage of time spent in the work sharing queue

fragmented, quality balanced queue – is superior to its simpler and presumably, less costly alternatives. Unfortunately, such implementations are not easily done, which reduces the much-heralded ease of implementation offered to application programmers by the multiprocessor's shared memory model.

Ease of programming is the topic of the remainder of this paper, where we describe an object-based programming library for the implementation of shared, distributed objects, called the DSA library. The library's functionality is stated in Section 3 using the 'tour' object from the TSP code as an illustrative example. The library's evaluation in Section 4 demonstrates its lightweight nature, and it shows how locality of access to such abstractions is improved by use of the library's primitives. In fact, we demonstrate similar performance for the TSP application with a library-supported fragmented queue to the queue's custom, shared memory implementation described in this section. Performance evaluation is performed on two different target machines: (1) a 32-node GP1000 BBN Butterfly multiprocessor and (2) a 32-node Kendall Square Research KSR-1 supercomputer. The library's portability to those machines is due to its implementation on top of a portable lightweight threads package developed by our group[33].

3 The DSA Library: Implementing Distributed Objects

The *DSA library* facilitates the implementation of shared, distributed abstractions in parallel programs in two ways:

1. at *user-level*, by providing application programs with a uniform interface to shared abstractions, and

2. at *representation-level* and *implementation-level*, by providing machine-independent mechanisms for the efficient and portable implementation of object fragments and access methods on the underlying parallel machines.

A library-constructed *DSA object* defines a communication structure – called a *topology* – and a communication protocol among object fragments represented as vertices of that structure

Object binding. Bindings are established, and broken using the following library routines:

```
TOP_RESULT top_open( obj_handle, obj_id, vertex_id );
TOP_RESULT top_close( obj_handle );
```

The 'top_open' routine returns a handle for future accesses to the specified object's vertex (i.e., the vertex number 'vertex_id' of the object instance identified by 'obj_id'). Vertex specification is necessary since it is possible to map multiple vertices of an object to the same processor node (ie., a vertex plays a role somewhat similar to a 'context' in Nexus[14]). An error status is returned if the specified vertex is not located on the calling thread's node. The 'top_close' routine breaks the binding associated with 'obj_handle', but it does not 'clean up' object state for future use. Such cleanup has to be implemented by additional operations called explicitly by application programs.

Object invocation. A thread can invoke any bound vertex's operations, using one of the following four library routines:

```
TOP_RESULT top_send( obj_handle, srv_id, param, param_size, tag );
TOP_RESULT top_send_w( obj_handle, srv_id, param, param_size, tag );
TOP_RESULT top_receive( obj_handle, srv_id, param, param_size, tag );
TOP_RESULT top_receive_w( obj_handle, srv_id, param, param_size, tag );
```

The effect of 'top_send' is the invocation of the service identified by 'srv_id' in the vertex identified by handle 'obj_handle'. Invocation parameters must reside in parameter block 'param', where 'param_size' indicates the block's size. Each invocation may also specify an arbitrary, user-provided 'tag' value, which may be used for communication of sequencing information, etc.
If user programs require synchronization with output generation at the local vertex, they may invoke the vertex operation 'top_send_w'. This operation will block the invoker until the invoked service and vertex have generated all of the required outputs.
A user thread obtains the result of a service executed by an invoked vertex by calling 'top_receive'. This routine copies the parameters returned by service 'srv_id' into the buffer pointed to by 'param'. The 'tag' parameter permits a wild card value.
The 'top_receive_w' routine blocks the caller thread until the requested return value is available at the local vertex. Since such threads resume execution in the 'top_receive_w' routine, they will complete the receive upon being dispatched.

Figure 6: User Level: Object Binding and Invocation

```
void new_tour( obj_handle, new_value )
top_h obj_handle;
int new_value;
{
  top_send( obj_handle, NEW_TOUR, &new_value, sizeof( int ), 0 );
}

int read_tour( obj_handle )
top_h obj_handle;
{
  int tour_value;

  top_send( obj_handle, READ_TOUR, &tour_value, sizeof( int ), 0 );
  top_receive( obj_handle, READ_TOUR, &tour_value, sizeof( int ), 0 );
  return tour_value;
}
```

Figure 7: User level: Procedural Interface of the "tour" Object

and interacting with user threads. Uniformity implies that to such user threads, a fragmented DSA object appears as a single abstraction shared among them. This is attained by having each fragment of the object export the same operations that may be invoked by any thread able to access it. We call this the *user-level view* of a shared abstraction. In contrast, to attain efficiency in implementation, the implementor of a DSA object (viewing the object at *representation-level*) may fragment the object's representation into the aforementioned set of connected object vertices, where different vertices (1) may be stored in different memory units, and (2) must explicitly communicate with each other in order to execute some (or all) of the operations performed on the object. Object vertices, then, must be implemented to jointly and cooperatively execute the object's operations and store and maintain the object's internal state (i.e., the *implementation-level view*).

3.1 Object Binding and Invocation

At *user-level*, the DSA library offers routines for binding a user thread to an already defined object and for invoking the object's operations. These routines are described and discussed in Figure 6, where we list the library's calls with which user threads first bind themselves to some fragment of the shared abstraction using the 'open' call, then invoke its methods using 'top_send' and 'top_receive' calls, and finally, 'close' the binding when they no longer need it. In DSA programs, each vertex may be bound to zero or multiple threads, and each thread may be bound to multiple vertices of the same or of different DSA object instances. However, the current version of the library requires that a binding is performed only between a thread and locally stored vertices.

Object implementors can use the 'top_send' and 'top_receive' calls explained in Figure 6 to create procedural interfaces to DSA objects. A sample procedural interface is shown for a shared 'tour' object in Figure 7, where the 'new_tour' operation updates the shared tour object with a new tour value discovered by the application thread. The 'read_tour' operation retrieves the

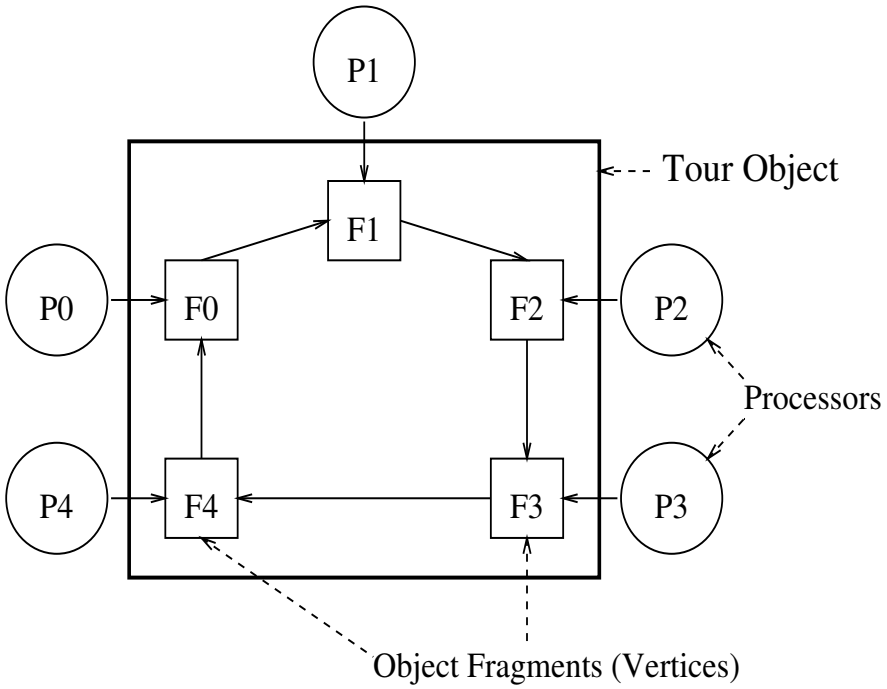

Figure 8: The Tour Object

current tour value from the shared object. This operation first executes the 'READ_TOUR' service by performing a 'top_send' operation, then retrieves the result of that service using the subsequent 'top_receive' call. 'Top_send' inserts the the local fragment's (called a *vertex*) best tour value in the vertex' output queue. The inserted value may be propagated to other object fragments, as per the consistency policy implemented by this shared abstraction, and it may be retrieved by local threads using the subsequent 'top_receive' call shown in 'read_tour'. Therefore, in our implementation of the DSA library, the execution of 'top_receive' cannot cause the execution of services. Services are executed only in response to 'top_send' operation. This simplification resulted in performance increases for the 'top_send' and 'top_receive' calls described in more detail in Section 4.

Finally, to an end user, the object fragment shown in Figure 8 appears much like a 'proxie'[45], since it locally emulates the object's complete functionality by exporting all of its operations. Namely, end users only know that the object's fragments 'F0' to 'F4' reside on processors 'P0' and 'P1' and that this tour object offers the operations 'read_tour' and 'new_tour' and contains the current, global 'best_tour' value as private data.

3.2 Object Creation

As apparent from Figure 8, a shared abstraction like the 'tour' object is represented as a set of multiple object fragments potentially located on different processors and connected via a statically defined logical communication structure. Such information is described at the library's *representation-level*. Again consider the tour value shared by all searcher threads in the TSP application. For simplicity, we represent this object as identical vertices 'F0' to 'F4'[2] linked by a ring communication structure (on distributed memory machines, alternative communication

[2]The use of different versions of object vertices in a single DSA object is not supported in the current implementation of the DSA library. Such a generalization of DSA objects can be useful, and is discussed further in [41].

structures using hardware-supported broadcast instructions or broadcast trees are commonly used). Figure 8 depicts the directed graph representing the object's internal communication structure and the vertices representing the object identical fragments. Communications among vertices are not visible to object invokers. In the case of the tour object, such communications concern updates to the local copies of 'best_tour' values stored in object fragments. Specifically, in this implementation (see Figure 7), both 'read_tour' and 'new_tour' can initiate the propagation of a new tour value around the ring to other object vertices. This propagation may be performed synchronously or asynchronously to the execution of additional operations on the local or remote object vertices, so that the desired consistency of the multiple copies of tour values around the ring can be controlled by the tour object's implementation.

At *representation-level*, the DSA library offers routines for implementing individual object vertices, including their operations, their communications with other vertices, the mapping of object vertices to processors, etc. This is described in Figure 9, where we define the implementations of the 'new_tour' and 'read_tour' operations exported by the tour object as consisting of the two *service routines* 'new_tour_srv' and 'read_tour_srv'. Pre- and post-conditions may be associated with calls to these routines (explained in more detail in Section 3.3). Temporary and permanent state used by these service routines and resident in object fragments is stored in a data structure of type 'ADT' (explained in the following section).

In Figure 9, following the specifications of services are the descriptions of the communication structure, the mapping of the fragmented object to the underlying parallel machine, and the call used for object creation. The creation of DSA object instances is typically performed at the time of program initialization. Once created, an object instance cannot be removed until program termination. Furthermore, when creating an object instance, an application has to describe the DSA object in its entirety, as evident from the details of the 'top_create' call shown in the Figure.

3.3 Object Fragments

At *implementation-level* and in the heart of the implementation of the DSA library's support for distributed objects are mechanisms for the implementation of object fragments, services, conditions, etc. These mechanisms are explained in this section.

The key insight concerning the implementation of DSA objects is the realization that a user thread's invocation of an object fragment may occur at a time different from the receipt of communications from other fragments. As a result, it must be possible for object implementors to explicitly describe the conditions under which object fragments' services execute with respect to the receipt of communications from other fragments and/or invocations from end users. To permit such asynchrony, the DSA library maintains queues at the input and output edges of vertices (as indicated in Figure 10), and it permits implementors to state *pre-* and *post-conditions* that can make the execution of a service conditional on fragment state (e.g., on the availability of inputs to the fragment). In addition, pre- and post-conditions may be used to implement asynchrony between service invocation and execution. This is particularly important when services are executed by their own threads. Pre- and post-conditions are stated separately from basic service functionality in order to improve service reusability.

Each service itself may be represented as an ADT (Abstract Data Type) or as a TADT (Threaded Abstract Data Type), so that a service may be executed synchronously with the receipt of an invocation from a bound thread (i.e., executed by the invoking thread (ADT)),

Services. The two operations of the tour object are implemented by routines 'read_tour_srv' and 'new_tour_srv', respectively. These routines and any required local state (e.g., local copies of tour values) are stored with each vertex. The following *services table* is located in each vertex of the tour object:

```
static top_srv_s tour_srv_table[] = {
  {READ_TOUR, read_tour_precond, read_tour_srv, read_tour_postcond, ADT },
  {NEW_TOUR, new_tour_precond, new_tour_srv, new_tour_postcond, ADT }
};
```

Topology. The logical communication structure of an object is described as a $N \times N$ from/to connection matrix, where N is the number of vertices. For instance, the structure of an object that has 8 vertices connected by a ring may be described as follows:

```
static int connections[][] = {
  { 0, 1, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 1, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 1, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 1, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 1, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 1, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 1 },
  { 1, 0, 0, 0, 0, 0, 0, 0 }
};
```

A value of '0' states that no link exists between two vertices, whereas a value of '1' represents a unidirectional link between two vertices (i.e., an edge). The vertices of an object are numbered from '0' to 'N-1'. A specific vertex is identified by its 'vertex_id', which is the vertex number. Routines able to generate such a matrix at the time of program initialization[44] may be used in place of the simple statically defined structure shown in this example.

Mapping. The mapping of vertices to physical nodes is described by a table with 'N' entries. The indices into this table are the vertex numbers, and the table elements are physical node numbers. For example, a one-to-one mapping of the ring structure shown above is described by the following table, but is typically computed by initialization-time routines:

```
static int mapping_table[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
```

Creation. Given the matrix and table structures shown above, an application creates a mapped object instance by calling:

```
TOP_RESULT top_create( object_id, size_of_private_data,
                       nb_of_services, services_table,
                       nb_of_vertices, connection_matrix,
                       mapping_table, nb_of_free_ib, max_param_size );
```

This routine returns a unique instance identifier, called an 'object_id'. The first seven parameters describe the new object's id, the space required for each fragment's state, the number of service routines whose addresses appear in the 'services_table', the number of vertices and the connection matrix, and the mapping of vertices to physical processor nodes. The last two parameters determine the size of the pool of pre-allocated invocation blocks associated with each object fragment.

Figure 9: Representation level: Creation of a Sample DSA Object

or it may be executed by a separate thread scheduled in response to changes in truth values of preconditions (TADT). In summary and as shown in Figure 9, the components of a fragment's service are (1) a unique identifier, (2) three procedure addresses, including (a) a procedure performing *precondition* evaluation, (b) a procedure implementing the actual operation, called a *service routine*, and (c) a procedure performing *postcondition* evaluation, and (3) a representation specifier. Each of these components is discussed in detail below.

Pre- and Post-conditions. The successful execution of certain DSA object operations may depend on their invocation by several user threads on different processors, and it may depend on the successful forwarding of information between different object fragments. As an example, consider the computation of a global sum using a 'combining tree' for the incremental collection and addition of individual threads' contributions to the sum[41]. Here, a service in a vertex at a certain level in the combining tree cannot be scheduled for execution until its bound thread has performed its invocations (i.e., contributed its partial sum) and until tree nodes at the lower levels of the combining tree have contributed their partial sums. This example demonstrates that one general role of preconditions is the definition of a service scheduling policy, based on the availability of inputs for that service in a particular vertex. For instance, a precondition may require that inputs from all input edges must be present in order to activate a service, as shown useful in synchronization objects implemented as combining trees or in certain implementations of objects implementing global sums or minima.

Postconditions associated with service routines can control the propagation of values across the object's communication structure by controlling output generation at vertices. An output may be generated after each service execution, or after some delay required or desired by the application. Furthermore, the result of a service's execution may be sent to one, some, or all output edges of a vertex, or to a user thread requiring it. For example, in a tree-structured global sum object, while each vertex can incrementally perform its addition operations upon the arrival of each input, each single output cannot be generated until all inputs have been received and added. This requires the use of a postcondition. By default, a DSA service routine is activated incrementally as each input for that service arrives.

In summary, the purpose of the precondition and postcondition procedures executed with each object invocation (if such routines have been specified) is to determine (1) when a service routine is activated in response to an invocation (service scheduling), (2) when control is returned to the user thread (invocation control), and (3) what, if any, other object fragments must be accessed for execution of the desired service (fragment management). In the tour object, a 'new_tour' operation has no precondition, but its postcondition states that control can be returned to the user thread after the local copy of the value has been read. In addition, the postcondition routine automatically initiates the execution of the service routine associated with the 'new_tour' operation on all other tour fragments. This is performed by invoking the next fragment along the ring, which in turn invokes its neighbor, etc. Such updates terminate when the postcondition routine detects that its local copy was equal to the received value.

Input and output queues. It is apparent from the discussion above that each object vertex is constructed such that its operations (services, pre- and postconditions) can be executed asynchronously with the invoking program. Furthermore, a vertex' services may be executed in response to invocations from other fragments or from a locally bound user-level thread. As

a result, each fragment's implementation contains addressing information about bound threads and connected vertices, and it contains several queueing structures in addition to the aforementioned object state and its user-specified services and pre- and postconditions. These queues shown in Figure 10 for a sample object fragment with three input edges, three output edges, and any number of bound threads include: (1) an *input queue* shared by all threads bound to the vertex, (2) an *edge queue* for each edge providing input to the vertex from other vertices (two such queues exist for each 'tour' fragment, one for its inputs from another vertex, one for invocations by the single bound user thread), and (3) output queues for each vertex output. Two such output queues should exist for the 'tour' vertex, one for the outputs generated for invocations from user threads, another for outputs to the vertex to which it is linked. An optimization in the shared memory implementation of the DSA library is the elision of all explicit output queues linking vertices to each other. Instead, each vertex uses as its output queues the edge queues of the vertices to which it is linked.

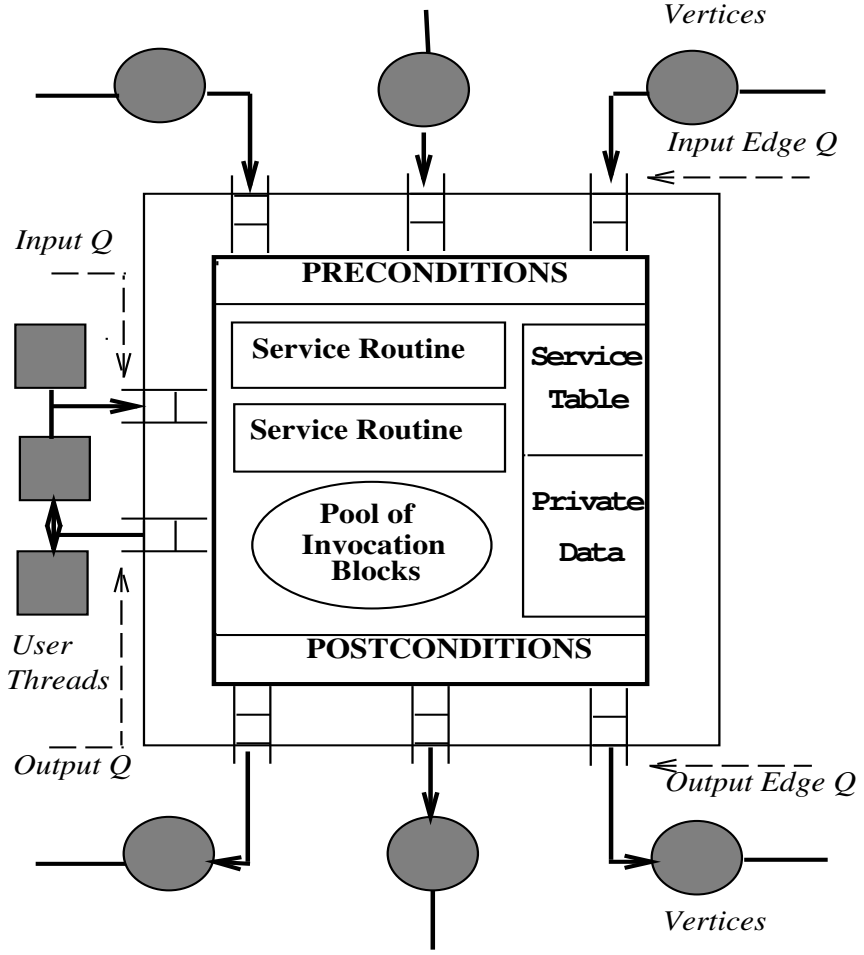

Figure 10: Object Fragments

Service representation. Since services may range from simple, low-latency message switching

to complex computations, the DSA library offers two different execution modes for service routines:

- ADT – small grain computations can be performed by service routines implemented as procedures called in response to an invocation. Execution of such a service is atomic (non-preemptible), and multiple invocations of it are thus implicitly serialized.
- TADT – larger grain computations can be performed by service routines represented as preemptible threads. A new thread is created for each invocation of such a service. Explicit synchronization is required for protecting a vertex's private data. Threads executing service routines are scheduled in a round robin fashion. Since application threads may block waiting for a service routine to be completed, threads executing service routines have priority over user-level threads.

Additional detail on DSA library support for service routines appears in Appendix A.

Service routines. Service routines perform the computations implementing an object's operations, and they are executed in response to fragment invocations by attached user threads or in response to message receipts from neighboring fragments. In either case, the information required for service execution is contained in an *invocation block* queued in the fragment's input queue. Each invocation block contains routing information (source and destination vertices), an identifier of the invoked service, a buffer into which the parameters required by this service have been packed, and a tag value. The invocation block only contains a pointer to the actual parameters, so that unnecessary copy operations are avoided. Detailed examples of service routines, using postconditions and implementing some application-dependent notion of memory consistency are demonstrated for the 'tour' object in Appendix A.1 and in Figure 18.

Remote invocations. The edges connecting object fragments are uni-directional, logical communication links. While the physical representation of such an edge is the appropriate edge queue of the target vertex, all communications across edges use a remote fragment invocation mechanism. As an example, consider a link from vertex *v1* to *v2*. Whenever a service routine in *v1* outputs a new tour value across this edge (i.e., enters data into the appropriate edge queue of *v2*), it also initiates the execution of the target vertex' service routine 'new_tour_srv'. The resulting remote queue access coupled with remote service routine execution comprises the remote invocation protocol used by the library for fragment communications. Such remote invocations can be *immediate*, which means that the control flow on the target vertex' processor is interrupted (using Unix 'signal' operations), or they can be *delayed*, which means that the remote service will be executed only when the user thread bound to the remote vertex executes one of the vertex' operations. Both alternatives have been implemented and are evaluated in Section 4.

4 Evaluation of the DSA Library

The DSA library provides support for the implementation of efficient fragmented objects on multiprocessor platforms. Its evaluation must demonstrate:

- the library's *utility* for constructing fragmented objects exhibiting high performance, and

- the library's *generality* in terms of its use on different target platforms, and its use with a wide variety of objects, including 'small' objects accessible with low latencies.

In this section, library *utility* is demonstrated by presenting (1) the low costs of the implementation of its basic constructs, and (2) the high performance attained by application programs using objects constructed with the library. Specifically, we use the DSA library to construct a distributed queue object that replaces the custom, shared memory distributed queue implementation used with the TSP program and evaluated in Section 2. We will show that the TSP program's performance with the DSA queue object is similar to its performance with the object's custom implementation. In addition, we describe some interesting tradeoffs between different implementations of the library's low level mechanisms. These tradeoffs concern the use of active messages vs. polling for remote fragment invocation.

The DSA library's *generality* is demonstrated by its use on two different target parallel machines, 32-node GP1000 BBN Butterfly and KSR-1 multiprocessors. Additional implementations existing for SGI multiprocessors and for distributed systems[27] are not evaluated in this paper.

Descriptions of target hardware. The GP1000 BBN Butterfly used in this research is a MIMD, shared-memory parallel machine, where each processor node contains a 25Mhz Motorola MC68020 processor, a 68881 floating point processor, a 68851 Memory Management Unit (MMU), 4M bytes of RAM, and a microcoded co-processor called the Processor Node Controller (PNC) which handles shared memory requests. Processor nodes are connected by a 32 megabits per second per path multistage switch which allows processor nodes to share their local memories with other nodes. For reference, a procedure call without parameters costs approximately 3 μseconds on the BBN Butterfly, a call to a local abstract data type (an ADT) costs about 18 μseconds, and a thread context switch in our lightweight threads library costs about 215 μseconds.

The KSR-1 supercomputer is a NUMA (non-uniform memory access) shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes or 34 rings (the largest machine delivered to date consists of 256 processors). Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache, a higher performance 0.5 MBytes sub-cache, and a ring interface. CPU clock speed is 20 MHz, with peak performance of 40 Mflops per node, an access time to the subcache of 2 processor cycles (with a 64-byte cache line), an access time of 18 processor cycles to local memory, and an access time of 126 cycles to remote memory using a 128-byte cache line. Therefore, severe penalties exist concerning accesses to sub-cache, cache, and remote memory. Such penalties increase when additional rings exist in the memory access hierarchy. Programmers do not perceive the memory hierarchy existing in the machine (other than by potentially observing performance penalties). Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap[18, 35]. For reference, a procedure call without parameters costs approximately 1.8 μseconds on the KSR-1, a user-level thread fork using our Cthreads library costs about 71 μseconds, and a thread context switch in our lightweight threads library costs 38 μseconds.

4.1 Basic Costs

Efficiency concerns have resulted in the optimization of frequently used internal library code and data structures, which can be described in terms of three layers: (1) object access via library routines, (2) local service execution and remote service execution by interaction with other fragments, and (3) inter-fragment communication mechanisms. Evaluations of (1) - (3) are presented next.

Object representation and creation. The performance of DSA objects depends in part on their internal representation. In addition to the queueing structures used for fragment inputs and outputs, each fragment is referenced via lists maintained on their processors. Specifically, all vertices located on a processor are linked via a local vertex queue as shown in Figure 11.

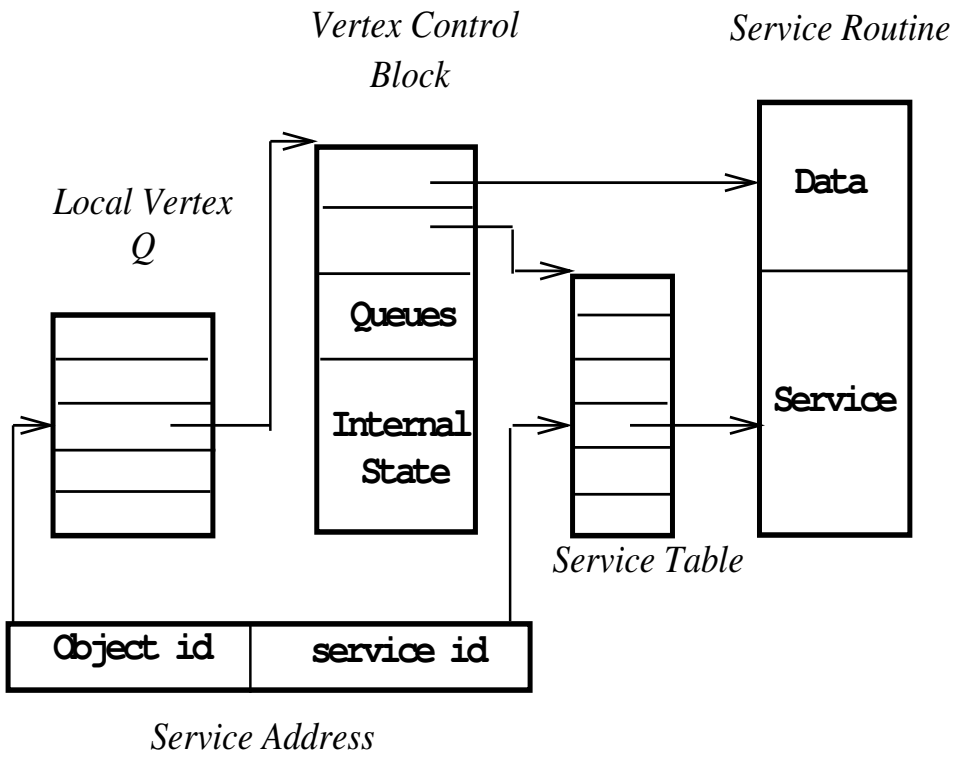

Figure 11: Addressing mechanism used in service invocation

As shown in the figure, each vertex is internally described by a vertex control block (abbreviated `vcb`). A `vcb` contains identifiers and several critical data structures, including (1) a private data buffer, (2) an input queue for the vertex input(s), (3) an output queue used by any threads bound to the vertex, (4) a waiting queue for threads blocked on services, (5) a pool of free invocation blocks, (6) a table describing the object's services, (7) a table describing the output edges (vertex id and node number of each linked vertex), and (8) an array of pointers to all of the object's `vcb`'s. The latter array permits a vertex to access any remote `vcb` of the object by direct reference.

Object creation has not been optimized in the current implementation, in part because

objects are typically created at the time of program initialization and are deleted only when the program terminates. However, it is instructive to consider the steps necessary for object creation and undertaken by the library routine 'top_create'. This routine first allocates each of the object's vcbs on the appropriate nodes according to the given mapping table. It then initializes these vcbs as per the object's description. Finally, 'top_create' sends a creation event with the appropriate vcb to each target node. Upon reception of a creation event, the event dispatcher calls a setup procedure, which enqueues the transmitted vcb in the local vertex queue. Given these steps, the performance of this call depends primarily on the performance of the threads library's calls for memory allocation (see [42]) and on the DSA library's calls for remote fragment invocation. The latter are reviewed below.

Object binding. A user thread binds itself to an object's vertex using the 'top_open' routine. This routine first performs a linear search for the vcb of the specified vertex on the local vertex queue. It then allocates a user control block and binds the calling thread to the specified vertex by storing the thread identifier and the vcb address in the user control block. As an optimization, the latter also contains a single invocation block for use in 'top_send' and 'top_receive' calls by the bound thread. The 'top_open' routine returns a pointer to the user control block as an object handle.

Object access. The performance of program using DSA objects critically depends on the performance of object access. We first consider 'top_send'. Its current implementation consists of four steps: (1) disable events, thereby preventing other operations on the local vertex while it is operating on it, (2) acquire and initialize the single invocation block 'owned' by the bound thread, which includes noting the service id, sizes and addresses of the call's parameters, (3) perform a local invocation of the requested service, and (4) enable events. Invocation parameters resident in the invocation block are directly accessed by the service; they are not copied out of the parameter block unless otherwise indicated.

The performance of 'top_send' operations on the GP1000 BBN Butterfly is depicted in Table 1 for representation of services as 'procedures' (abstract data types – ADTs) or as threads (TADTs)[3]. The latency of a simple 'top_send' operation with a no_op service demonstrates the basic overheads of invocation block manipulation and event disable/enable. When a post-condition associated with the service results in output being performed, call costs increase due to output queue manipulation. The table also depicts the additional costs arising from a call of 'top_output_user' with a single invocation block. The cost of allocation for a single remote invocation block is 43 μseconds.

Operation	ADT	TADT
top_send	109	912
top_send + top_output_user	232	1020

Table 1: Costs (μseconds) of 'top_send' operations

When 'top_send' is performed for services represented as threads (TADTs), additional context switch overheads arise, since the invoking thread has to release the processor, followed by

[3]These measurements are attained on a single processor node, using the average latency derived from 1000 consecutive calls.

the processor's acquisition by the thread executing the service. Two alternative implementations of such context switching on the BBN Butterfly (1) use an un-optimized operating system call that saves signal masks vs. (2) use an optimized context switch for lightweight threads. The costs of (1) are 2.5 milliseconds on the BBN Butterfly, whereas (2) requires 215 μseconds. A service invoked in 'delayed' mode (i.e., assuming that any required remote fragment's services will poll for inputs) can take advantage of the low costs of (2), whereas (1) must be used when using a user-level implementation of active messages for remote service invocation on the BBN Butterfly (i.e., invocation in *immediate* mode) using Unix signals rather than using the low-level interrupts available at kernel level. This results in unacceptable latency for invocation of threaded, immediate services, vs. the 912 μsecond cost of service invocation for threaded services in delayed mode shown in Table 1.

The 'top_receive' routine executes the following four steps: (1) disable events, (2) perform a linear search for the requested invocation block in the vertex's output queue and remove it, (3) copy the parameters stored in the invocation block to the user provided buffer, and (4) enable events. 'Top_receive_w' executes the same four steps; in addition, it blocks the caller thread if the invocation block is not present in the output queue, as explained in Appendix A.1. As apparent in Table 2, the performance of 'top_receive' is same for ADT and TADT services since service execution is not performed in response to 'top_receive'.

Operation	ADT	TADT
top_receive	123	123

Table 2: Costs (μseconds) of 'top_receive' operations

These timings assume that the required invocation block is available, and that there are no other threads waiting to receive from the vertex being invoked.

In summary, it is apparent from the measurements presented above that the cost of accessing a shared abstraction's local fragment are moderate, resulting in acceptable performance (i.e., overheads of 20% or less) for abstractions with service execution times exceeding 500 μseconds. As shown in Sections 4.3 and 4.4 below, despite these overheads, the use of DSA abstractions with the TSP application results in performance gains similar to those attained with the custom queue implementation.

4.2 Remote Service Invocation

The DSA library uses remote invocation as an inter-vertex communication primitive. While low cost of remote invocation is critical to the performance of fragmented objects, the use of remote invocation vs. remote access provides several performance advantages on NUMA multiprocessors: (1) it tends to improve the locality of reference of programs by removing remote references, and (2) it provides implicit synchronization for cooperating threads.

The tradeoffs between remote invocation and remote memory access are empirically evaluated on the BBN Butterfly in [6, 8]. Results reported by the authors show that the overheads associated with explicit synchronization and remote references increase with increasing 'sizes' of remotely accessed data and code (i.e., with the complexity of remote operations), whereas the overheads associated with remote invocation do not depend on those sizes. Therefore, remote

references outperform remote invocation only on 'simple' operations. Such results are part of our motivation for implementation of the low-level remote invocation construct for inter-vertex communication.

In DSA, a remote invocation is initiated by calls to the 'top_output_edges' or 'top_output_vertex' routines. The resulting remote service invocation is similar to active messages, but is actually derived from the active message mechanism employed in our earlier work on distributed objects on hypercube machines[41]. A remote invocation is comprised of the following steps: (1) extract a free invocation block from the target vertex' pool using remote references, (2) copy the invocation data into this block, including parameters, and (3) send a request containing the invocation block to the target fragment and processor. Such a request is described by an *event*, which defines an asynchronous action that is to be performed on the remote processor's fragment. When processing such an event, the target *event dispatcher* associated with that fragment (1) performs a local invocation of the appropriate service, and upon its completion, (2) places the invocation block back into the pool of its home vertex.

The performance of remote fragment invocation depends on the costs of event transmission via an event transmission facility and the costs of event activation at the target in either *immediate* or *delayed* service modes mentioned earlier. In order to reduce the costs of event generation, each processor locally maintains an event queue and a pool of pre-allocated event descriptors. These two data structures are protected by a spin lock. In the shared memory implementation, the message associated with the event is transmitted by reference using a low-level mailbox communication facility. In the distributed memory implementation described in [41], event transmission employed modifications of the low-level communication protocol on the iPSC hypercube. The implementation described in [27] sends messages between daemon processes when events are transmitted.

Operation	Cost
event generation	66
sending an event	187
event handling	153
total	**406**

Table 3: Cost Breakdown (μseconds) of a remote invocation in delayed mode

As listed in Table 3, the total cost of a single remote invocation is 406 μseconds in delayed mode. *Delayed mode* implies that the remote invocation is not executed until the target fragment checks for its event's existence, then picks it up and executes the event. In *immediate mode*, a Unix signal is generated when the event is transmitted, which interrupts the target processor and prompts it to execute the event immediately, resulting in an execution time (not elapsed time) of 1.15 milliseconds on the BBN Butterfly, of which more than 700 μseconds are due to the cost of Unix signal generation. Both of these measurements are attained with an invocation block containing a two byte parameter. A kernel-level implementation of remote invocations in immediate mode like the one described in [6] (using hardware interrupts instead of Unix signals) would reduce the cost of immediate remote invocations to roughly 500 μseconds on the BBN Butterfly. Such an implementation would also remove the second obstacle on the BBN machine for attaining high performance for immediate invocations, namely, the total elapsed time for

invocations in immediate mode depends not only on the time required to generate an event and signal but also on the time required to deliver this signal. On the GP1000 BBN Butterfly, signal delivery has been observed to vary from 1 to 110 (!) milliseconds by our group.

It should be apparent from the discussion in the previous paragraph and from the discussion addressing Table 1 that the performance of *immediate events* using Unix signals is not satisfactory. For instance, consider an asynchronous DSA object resembling the tour object in the TSP application. This object links one thread to itself with a 8 vertices ring spanning 8 nodes. The evaluated service performs only routing of incoming invocation blocks. Due to the extreme variability of Unix signal delivery times, we have measured total round trip times ranging from 18 to 400 milliseconds for invocations traversing this ring.

Due to the high costs of event generation and delivery in immediate mode, the DSA library's BBN Butterfly implementation employs several optimizations of event transmission and servicing. First, since event generation is expensive, several simultaneous events can be grouped into a single event, by simply generating a single event descriptor for multiple invocation blocks entered into the target vertex' input edges. Upon receipt of the event, the target vertex' service routine processes all invocation blocks found in the appropriate input edges. Second, the remote event queue is checked prior to signal generation. An empty queue implies that the remote vertex is currently running the event handler, so that a signal need not be generated. The third optimization concerns event masking. Specifically, signal generation requires disabling and enabling events when applications currently execute inside certain critical sections or when the target process is currently executing an event handler. Since such event handling is quite expensive on Unix systems (Unix signal masking/unmasking system calls cost 800 μseconds on the BBN Butterfly), our implementation maintains 'events enabled' and 'events disabled' flags on each processor. These flags are set by the local event handlers and inspected at the time of event generation. The event generation routines do no issue signals to the target vertex when its events are currently disabled, since that implies that the event handler is currently running on the target processor and will receive and process the invocation blocks that have already been generated and added to the appropriate input edge queues.

To summarize, the DSA library's implementation of event generation and delivery on the BBN Butterfly favors the use of simultaneous events and therefore, total event generation and processing overheads are reduced for increasing numbers of total events generated in the DSA abstraction's execution. In addition, delayed are preferable to immediate events due to the high cost and instability of Butterfly's Unix signal implementation and due to the library's increased portability with delayed events (we have experienced many 'minor' differences in Unix signal implementations on different parallel or sequential machines). We hypothesize that the availability of efficient operating system support for active messages would not change one basic insight derived from these measurements:

- *Delayed mode* invocations are most efficient – remote fragment invocations are most efficiently implemented by permitting the receiving fragment to poll for incoming messages in conjunction with user-level accesses to fragments. Such polling is possible when user-level access rates are high (e.g., for the shared queue in the TSP application).

- *Immediate mode* invocations are necessary – active message implementations of remote invocations are important when fragment access rates are low or differ widely across fragments (e.g., as in the shared tour example), since they permit a fragment to participate in communications even when its bound threads are inactive or do not use it. They

also enable the fragment to interrupt its bound threads, perhaps to discontinue useless computations being performed (e.g., when a new tour is found by another fragment's thread).

Both implementations meet the paper's original goal of improving the locality and reducing the contention of access to shared abstractions.

4.3 Performance of TSP with DSA Objects

This section provides additional demonstrations of the utility of the DSA library by evaluating its use with the TSP application. The first set of measurements reported below compare the performance of TSP's *variant distributedQB* when using the custom, shared memory implementation of the distributed queue vs. using the DSA library and *immediate mode* for implementation of the same queue variant (see Figure 12). Despite the significant overheads of event generation and handling experienced with this signaling implementation of DSA, performance results indicate that the DSA library is suitable even for larger-scale parallel systems: good speedup is attained when using the library.

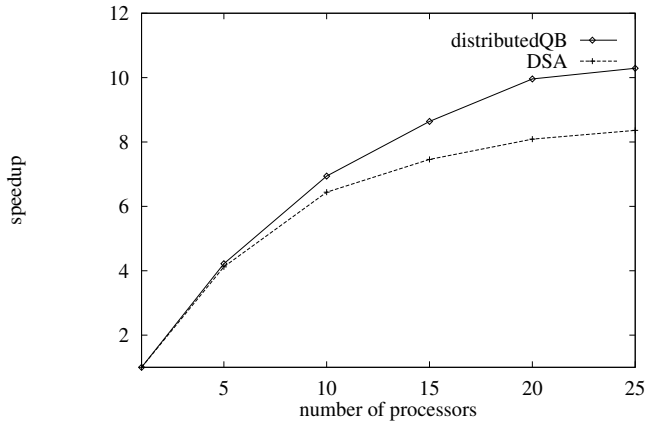

Figure 12: TSP Execution Times with or without the DSA Library

The speedup results depicted in Figure 12 are explained with additional measurements shown in Figure 13. This figure depicts the ratio of time spent in the work sharing abstraction vs. the application's total execution time. It is apparent from the observed ratios that the cost of DSA object use is roughly three times higher than the cost of using the direct shared memory implementation of queue variant distributedQB (due to the high cost of signaling in the BBN Butterfly's Unix implementation). However, some compensation for those additional costs arises from increases in program locality. Specifically, searcher threads interact only with the locally stored vertices, and all operations on remote vertices are performed by event handlers on remote processors.

Figure 14 demonstrates improved performance of DSA compared to the results shown in Figure 12) by eliminating the overheads of signaling. In these measurements, event activation on remote processors is performed in *delayed* mode. This means that no signal is generated

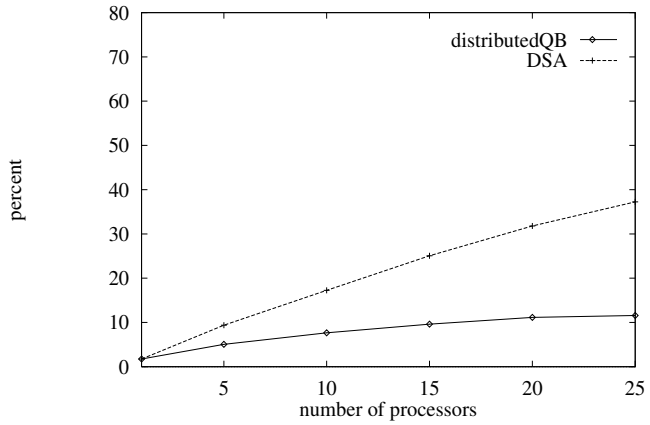

Figure 13: Percentage of time spent in the work sharing queue

when an event is entered in a remote event queue. Instead, the event queue is checked (polled) each time a local thread accesses the fragment (i.e., performs an operation on the fragment) and at that time, all events found in the queue are processed in arrival order. This polling approach works well for frequently accessed abstractions; it does not work for abstractions with vertices that are not bound to local threads (intermediate vertices used for communication only) or for abstractions that exhibit widely varying access frequencies to different fragments. Interestingly, the performance improvement seen in Figure 14 is not as significant as expected due to the various optimizations we performed on the signalling version (e.g., event batching, event flags, etc.).

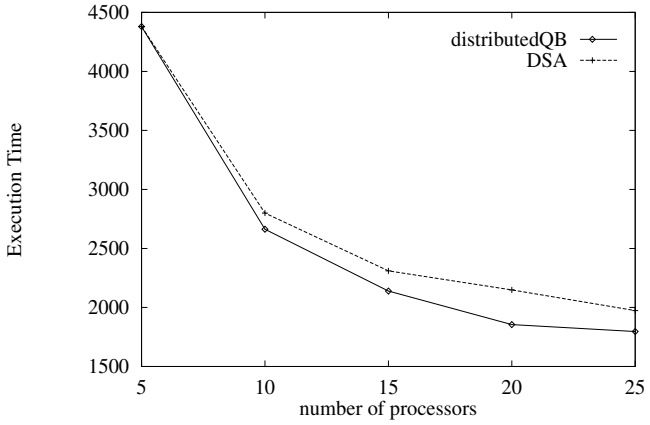

Figure 14: Execution times (milliseconds) of variants of the TSP application

4.4 Generality of the DSA Library

The most interesting observation from the KSR measurements shown in this section is that the sequentially consistent memory model offered by the KSR machine does not make it unnecessary for programmers to use libraries like the DSA library. This validates a basic premise of our research. Namely, any larger-scale parallel machine exhibiting NUMA memory properties must be used in a fashion similar to distributed memory machines, including the explicit distribution of the state and functionality of programs' shared abstractions. We next review KSR-1 performance measurements, followed by their discussion and interpretation in light of this paper's results and insights.

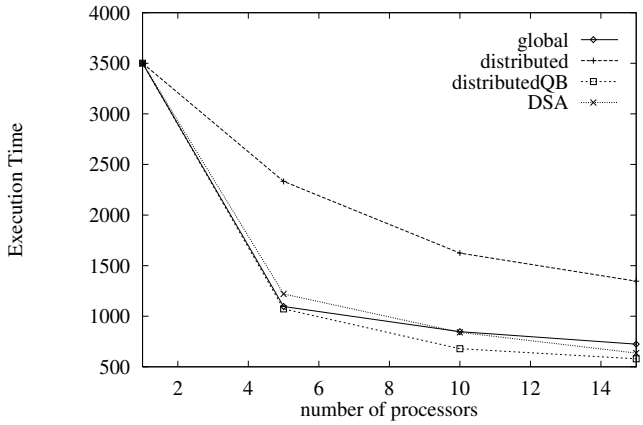

Figure 15: Execution times (μseconds) of variants of TSP on the KSR-1

Measurements of DSA performance on the KSR platform indicate results similar to those attained on the BBN Butterfly. Specifically, when using delayed event generation (i.e., event polling rather than signalling), we have shown that parallel programs written with the DSA library can deliver performance improvements for larger-scale parallel applications. Measurements of execution times achieved for the TSP application with the DSA library on a 32-node KSR-1 multiprocessor are shown in Figure 15. Actual execution times are comparatively smaller to those on the BBN Butterfly due to the KSR's faster processors.

Figures 16 and 17 show additional measurements on the KSR-1 machine, depicting the total number of nodes expanded in order to arrive at a solution, and the total percentage of time spent in the work sharing queue, respectively. These results are attained with an un-optimized implementation of the DSA library for the KSR machine. They generally mirror the results obtained on the BBN Butterfly machine. The remainder of this section discusses the impact of distribution on the BBN Butterfly vs. the KSR-1 machine. Toward this end, we next briefly describe the KSR machine's memory model.

KSR's ALLCACHE memory consists of a collection of local caches and a search engine that interconnects the local caches and provides routing and directory services for them. ALLCACHE implements a "sequentially consistent" shared address space programming model. Unlike the BBN Butterfly multiprocessor's memory architecture, data moves to the point of refer-

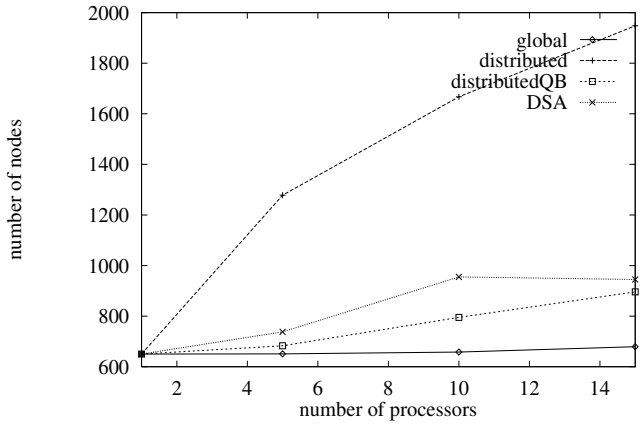

Figure 16: Total number of node expansions (KSR-1)

ence on demand. The underlying consistency protocol allows data replication (a read request gets a copy of the data in the local cache), and implements invalidation of all copies on a write request.

Given the ALLCACHE memory model, it should be clear that a centralized implementation of the shared queue is likely to be as inefficient on the KSR machine as on the BBN Butterfly, in part because such an implementation causes frequent cache invalidations and misses, resulting in significant amounts of data movement. This in turn results in high access ratios of remote to local memory for the TSP program. Given the extensive performance penalties for remote vs. local memory accesses on the KSR machine (worse than those on the BBN Butterfly), centralized queue implementations cannot be recommended for programs that frequently access such queues. This is particularly true for the TSP programs, where each searcher process tends to both write and read its own queue fragment, thereby causing it to be resident in its local cache.

Similar performance penalties can arise for queues used by one producer and many consumers, even when their elements are across different processor caches. This is because queue descriptions will tend to move from one cache to another depending on the access pattern to the queue. As a result, the DSA library should link any one of its vertices to several other vertices (i.e., inter-vertex communication) using multiple rather than single mailboxes. In addition, the performance of mailbox communications among each single producer/consumer pair (a unidirectional link between two vertices) may be improved further by using the KSR's `poststore` instruction, which permits the producer to immediately update data contained in the consumers' caches. Other optimizations involve packing messages into one cache line (128 bytes) and the elimination of queue locks for statically allocated structures (so that a queue element can always be accessed atomically).

In summary, the KSR measurements outlined in this section demonstrate that hardware support for sequentially consistent shared memory does not obviate the use of the DSA library, supporting the explicit fragmentation of shared abstractions and resulting in increases in locality of access. We posit that future tradeoffs in processor vs. memory and cache speeds will make

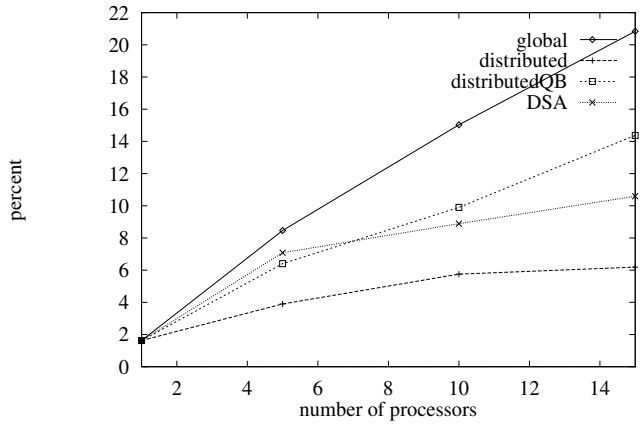

Figure 17: Percentage of time spent in the work sharing queue (KSR-1)

it increasingly important that programmers either use libraries like DSA or use the 'distributed memory' programming style it supports, even on small scale parallel machines like the SGI Powerchallenge multiprocessors. This hypothesis is borne out by initial measurements we have conducted on an SGI machine.

5 Related Research

The main topic of this paper is the efficient implementation of shared abstraction in NUMA multiprocessor programs, using a branch-and-bound solution to the traveling salesperson problem as a sample parallel application program. Previous work on parallel TSP includes that of Mohan in [32] employing the LMSK algorithm which we adopted for use with the DSA library. Finkel's distributed implementations of branch-and-bound algorithms[13] are difficult to compare to ours, because he analyzes the performance of TSP for alternative work distributions rather than for alternative methods of work sharing, for distributed termination, and for fault tolerance. Furthermore, his implementations carefully avoid the use of global knowledge. However, we share the notion of 'fairness' regarding work distribution among searcher threads with Finkel's work.

In [12], issues concerning the implementation of a best-first branch-and-bound algorithm on a hypercube multicomputer are discussed. Felten uses a structure called a 'decentralized queue' for storage of the nodes of the search tree. This structure is similar to our work sharing abstractions (variants distributed and distributedQB). Felten also points out the importance of ensuring that 'good' descriptions of work are scattered across the system. He proposes (but does not evaluate experimentally) a work sharing scheme similar to the one used in queue variant distributedQB in our research: to insert newly generated nodes into the local queue, but to send the locally second best node to a randomly chosen processor.

Other issues regarding parallel TSP implementations on hypercube machines are discussed in [40], as well as various alternative representations of the work sharing and tour abstractions on distributed memory machines.

There has been some earlier work on parallel program portability [10, 1, 9]. In [1, 9], authors have proposed portable models analogous to programming languages. Similar to our work, their approach is based on abstractions where interfaces and implementations of libraries are clearly separated. Based on the execution environment, an interface is bound to an efficient implementation. Although the DSA run-time system is based on a similar object oriented approach to increase portability, the focus of our work is quite different. While their emphasis is on portability based on models [1], late binding and program annotation [9] , our focus is on an active message style mechanism to implement object fragments to support efficient shared abstractions in parallel programs.

Related research regarding weak memory or fragmented memory in distributed systems[4, 46] has already been reviewed in the introductory sections of this paper.

6 Conclusions and Future Research

This paper presents the DSA runtime library for the efficient implementation of distributed shared abstractions in medium-scale multiprocessor systems. Measurements of the library's primitives and their evaluation with a sample parallel program on 32-node BBN Butterfly and Kendall Square multiprocessors demonstrate:

1. The DSA library supports the implementation of shared abstractions such that they are efficiently executable on medium-scale parallel machines. Scalability is achieved by explicit representation of such abstractions as multiple fragments located on participating processors.

2. The implementation of the DSA library assumes the availability of an efficient remote invocation mechanism used for communication among object fragments. The DSA library offers two implementations of this mechanism, one delaying the execution of a fragment's method until the fragment is accessed by a local thread, the other using active messages – resulting in immediate execution of a fragment's method asynchronously to the execution of other threads on the same processor.

3. The DSA library has been ported to and demonstrated efficient on multiple parallel machines, including a 32-node KSR supercomputer, the GP-1000 BBN Butterfly, and SGI multiprocessors.

The importance of the research presented in this paper derives from its exposition of object fragmentation as one important technique when implementing high performance objects. Our future research will address the topic of high performance objects with greater breadth. First, we will explore a variety of implementation techniques for high performance distributed and parallel objects for multiple target platforms. For example, we have already developed a first prototype of a framework for implementation of distributed shared abstractions and of distributed shared memory on networked machines[27]. In addition, on the KSR-2 multiprocessor platform, we have evaluated the extension of the underlying Cthreads packages with multiple, heterogeneous schedulers[34], so that one scheduler runs threads that perform event execution and a second scheduler runs application threads. Second, we will explore additional topics concerning the efficient support and implementation of distributed shared abstractions, including: (1) the customization and parallelization of network protocols for use in DSA implementations[30],

(2) the embedding of DSA objects in a more general system for the construction of object-oriented parallel programs[17], including the development of language-level and user interface support for object-based parallel programming, and (3) the interfacing of Cthreads and the DSA library with distributed programs written using existing general frameworks for programming networked, heterogeneous parallel machines, such as Nexus, PVM, or MPI. In general, the future platform for the construction of high performance distributed and parallel objects to be implemented as part of our research will offer support for both the construction and the tuning of objects, since the porting of existing objects to new platforms will probably be a more common activity than their initial construction. Third, we are developing application programs able to use such parallel and distributed machines, and we are constructing monitoring support for on-line program viewing and evaluation[11, 19].

The DSA library software is available in the public domain by remote FTP access to ftp.cc.gatech.edu.

References

[1] G. Alverson and D. Notkin. Program structuring for effective parallel portability. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1041–1059, September 1993.

[2] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 13(3), March 1992.

[4] J.K. Bennett, J.B. Carter, and W. Zwaenepol. Munin: Distributed shared memory based on type-specific memory coherence. In *Second Symposium on Principles and Practice of Parallel Programming, ACM, 23, 5*, March 1990.

[5] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[6] E.M. Chaves, Jr., P.C. Das, T.L. LeBlanc, B.D. Marsh, and M.L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–192, May 1993.

[7] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Dept. of Computer Science, Carnegie Mellon University, June 1988.

[8] A. Cox, R. Fowler, and J. Veenstra. Interprocessor invocation on a numa multiprocessor. Technical Report TR 356, University of Rochester, 1990.

[9] L. Crowl. *Architectural Adaptability in Parallel Programming.* PhD thesis, Department of Computer Science, University of Rochester, May 1991.

[10] D. Eager and J. Zahorjan. Enhanced run-time support for shared memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[11] Greg Eisenhauer, Weiming Gu, Thomas Kindler, Karsten Schwan, Dilma Silva, and Jeffrey Vetter. Opportunities and tools for highly interactive distributed and parallel computing. Technical Report GIT-CC-94-58, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, December 1994. Also in Proceedings of The Workshop On Debugging and Tuning for Parallel Computing Systems, Chatham, MA, October, 1994.

[12] Ed Felton. Best-first branch-and-bound on a hypercube. In *Third Conference on Hypercube Concurrent Computers and Applications, ACM*, Jan. 1988.

[13] R. Finkel and U. Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–255, Apr 1987.

[14] I. Foster, C. Kesselman, and S. Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical report, Argonne National Laboratory, 1995.

[15] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors*. Prentice-Hall, 1988.

[16] Kourosh Gharchorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[17] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[18] Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan. Experimentation with configurable, lightweight threads on a ksr multiprocessor. Technical Report GIT-CC-93/37, College of Computing, Georgia Institute of Technology, 1993.

[19] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, April 1994. Also in Frontiers 95.

[20] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[21] W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, D.A. Wallach, and W.E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[22] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[23] D. Sweeney J.D. Little, K. Murty and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11, 1963.

[24] Anita K. Jones and Karsten Schwan. Task forces: Distributed software for solving problems of substantial size. In *Proceedings of the 4th International Conference on Software Engineering, Munich, W. Germany*, pages 315–329, Sept. 1979.

[25] V. Karamcheti and A. Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing, Portland, OR.* ACM, May 1993.

[26] Carol Kilpatrick and Karsten Schwan. Chaosmon – application-specific monitoring and display of performance information for parallel and distributed systems. In *ACM Workshop on Parallel and Distributed Debugging*, pages 57–67, May 1991.

[27] Prince Kohli, Mustaque Ahamad, and Karsten Schwan. Indigo: User-level support for building distributed shared absractions. In *Fourth IEEE International Symposium on High-Performance Ditributed Computing (HPDC-4)*, August 1995.

[28] T. J. Leblanc. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986.

[29] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[30] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. In *International Conference on Network Protocols.*, 1993. TR# GIT-CC-93/22.

[31] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocssors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[32] Joseph Mohan. Experience with two parallel programs solving the parallel salesman problem. In *Proceedings of the 12th International Conference on Parallel Processing*, pages 191–193, Aug. 1983.

[33] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[34] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable micro-kernel. In *Proc. of Second workshop on Microkernels and Other Kernel Architectures*, September 1993.

[35] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59–66, July 1993. Also TR# GIT-CC-93/17.

[36] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: Kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, pages 259–270, september 1994.

[37] David M. Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of real-time distributed and parallel systems. Technical report, College of Computing, Georgia Institute of Technology, ICS-GIT-90/23, Atlanta, GA 30332, May 1990.

[38] M. Satayanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The itc distributed file system: Principles and design. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 35–50, Dec. 1985.

[39] M. Schroeder and M. Burrows. Performance or firefly rpc. In *Twelfth ACM Symposium on Operating Systems*, pages 83–90, Dec. 1989.

[40] Karsten Schwan, Ben Blake, Win Bo, and John Gawkowski. Global data and control in multicomputers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm. *Concurrency: Practice and Experience*, pages 191–218, Dec. 1989.

[41] Karsten Schwan and Win Bo. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

[42] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A c thread library for multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, GIT-ICS-91/02, Jan. 1991.

[43] Karsten Schwan, John Gawkowski, and Ben Blake. Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1520–1530, Jan. 1988.

[44] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Real-time threads. *Operating Systems Review*, 25(4):35–46, October 1991.

[45] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Sixth International Conference on Distributed Computing Systems*, pages 198–204, May 1986.

[46] M. Shapiro. Object-supporting operating systems. *TCOS Newsletter*, 5(1):39–42, 1991.

[47] Ellen Spertus and William J. Dally. Evaluating and locality benefits of active messages. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[48] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. Prelude: A system for portable parallel software. Technical report, MIT Laboratory for Computer Science, Technical Report MIT/LCS/TR-519, Oct. 1991. Shorter version appears in *Proceedings of Parle '92*.

A Library Support for Service Implementation

This section reviews the DSA library's low-level support routines used for service implementation. Toward this end, we first present some sample service routines in detail. Next, we describe some implementation aspects.

A.1 Sample Service Routines for a DSA Object

The tour object's operations 'read_tour' and 'new_tour' are implemented using the 'top_send' and 'top_receive' operations offered by the library, which in turn assume the existence of the service routines 'read_tour_srv' and 'new_tour_srv' as shown in Figure 18. ' **New_tour:** The routine 'new_tour_srv' is executed either as a result of invocation of the 'new_tour' operation on the local fragment, or when a message carrying a new tour value arrives at a fragment's input edge. If executed as a result of a message receipt from another fragment, the new tour value is contained in an invocation block 'ib' queued in the fragment's input queue. If executed as part of the 'new_tour' operation, the 'top_send' routine within 'new_tour' first acquires and fills an invocation block (ib) and then enters it in the bound vertex' input queue. Each invocation block contains routing information (source and destination vertices), an identifier of the invoked service[4], a buffer into which the parameters required by this service have been packed, and a tag value. This tag is an arbitrary value that may be used to identify a particular set of input invocation blocks. For instance, in a ring topology, the application code or pre-/postconditions can use the tag value to identify a previously sent invocation block that has fully traversed the ring.

Invocation block allocation is optimized in the tour object's implementation, by pre-allocating a single invocation block when a 'top_open' operation is performed on the local vertex. This single invocation block is used for all tour fragment invocations, thereby avoiding unnecessary invocation block allocations. However, the DSA library does not offer any general support for allocation optimization or for the optimization of parameter marshalling and the minimization of parameter copying, as done in other RPC implementations[39].

Service routines simply remove invocation blocks from the vertex' input or edge queues. For the tour object, whenever the 'new_tour' parameter is better than the locally stored value of 'best_tour', 'new_tour_srv' updates the local 'best_tour' value, and 'new_tour_postcond' enters the invocation block carrying the 'new_tour' parameter in the vertex' logical output queue (i.e., in the appropriate edge queue of the next vertex along the ring). Since propagation around the ring proceeds only in one direction, the best 'new_tour' values are fully propagated around the ring. Furthermore, the 'new_tour' service can propagate new tour values asynchronously to 'read_tour' operations performed on object fragments. Therefore, no guarantees are made as to the consistency of the tour values stored and read at different ring vertices at any one time. Experimental results on shared and distributed memory multiprocessors demonstrate that such inconsistencies only marginally degrade TSP performance.

'Read_tour': Reading tour values is a bit more complex. 'Read_tour_srv' is executed as a result of invocation of the 'read_tour' operation by a user thread, which in turn executes the 'top_send' and 'top_receive' routines. Let us ignore the 'top_send' since it is similar to the 'new_tour' operation described above. 'Top_receive' initiates the local execution of the appropriate service, which is 'read_tour_srv' in this case. The service routine 'read_tour_srv'

[4]The services defined in a DSA object are uniquely identified by an integer value.

```
void new_tour_srv( ib )
top_ib_t ib;
{
     int *local_best_tour = (int *)top_data_p( ib );
     int new_tour = *(int *)top_param_p( ib );

     if( new_tour < *local_best_tour )
     {
        *local_best_tour = new_tour;
        top_postcond( ib );
     }
}

void new_tour_postcond( ib )
top_ib_t ib;
{
     top_output_edges( ib );
}


void read_tour_srv( ib )
top_ib_t ib;
{
     int *local_best_tour = (int *)top_data_p( ib );
     *(int *)top_param_p( ib ) = *local_best_tour;

     top_postcond( ib );
}

void read_tour_postcond( ib )
top_ib_t ib;
{
     top_output_user( ib );
}
```

Figure 18: Sample Service Routines

removes 'ib' from the vertex' input queue and accesses all required parameters using the pointers stored in 'ib'. The service routine then copies the vertex's local 'best_tour' (stored in the local vertex' private data, termed 'top_data_p') into one of the operation's parameters. Next, the procedure 'read_tour_postcond' decides that 'ib' contains a value that should be made available to the invoking thread, and finally, the procedure 'top_output_user' returns the invocation results to the user thread by entering 'ib' into the vertex' output queue. 'Top_receive', then, simply scans the vertex' output queue for any invocation block matching the provided 'srv_id' and 'tag' arguments. In this case, such an invocation block is always available, and is promptly returned to the user for reuse in future invocations. If the postcondition routine is written such that output values may be witheld, then 'top_receive' may fail to find an appropriate 'ib' in the local fragment's output queue. In that case, 'top_receive' returns with an error status. Alternatively, if the user thread used the blocking variant of 'top_receive' ('top_receive_w'), the user thread is removed from the processor's ready queue and inserted into a queue of blocked threads associated with the vertex' output queue, noting the requested values 'srv_id' and 'tag'.

The potential existence of threads waiting for vertex output is another reason for the use of postconditions in the DSA package. When a postcondition generates vertex output to a user thread by calling the 'top_output_user' routine with a specific 'ib', the routine actually first enqueues 'ib' in the vertex's output queue, then scans the vertex's waiting queue for threads blocked waiting for 'ib'. If such threads are found, they are removed from the waiting queue, placed onto the processor's thread ready queue, and will eventually complete their 'top_receive' instructions.

A.2 Implementation Detail

Recall that a service is comprised of an optional precondition, the service routine, and an optional postcondition. We elide details of the implementation of invocation blocks and of the addressing information maintained in those blocks. Instead, we assume that such blocks are the atomic units manipulated at this level of the DSA library.

Preconditions. A local fragment invocation or an invocation from a remote fragment (using the library's remote invocation mechanism) initiates the execution of the appropriate service routine when there exists no precondition, else it calls the precondition procedure, in either case providing an invocation block (ib). The precondition is executed non-preemptively, and it must explicitly activate the actual service using the support routine:

```
void top_service( ib );
```

Activation of a service either involves calling the procedure defining the service, or creating a new thread that will execute this procedure, depending on the service's representation.

The aforementioned queueing structures inside each vertex are required because services can be implemented to execute asynchronously with the user threads requesting them. Precondition procedures can check and manipulate those queues using the routines:

```
void     top_enqueue_input( ib );
top_ib_t top_dequeue_input( service_id, tag );
bool     top_check_input( service_id, tag, condition );
```

'Top_dequeue_input' scans the vertex's input queue and dequeues the first invocation block that matches the given service identifier and tag value. The tag parameter admit a wild-card value.

'Top_check_input' checks the vertex's input queue for ib's that match the given parameters. 'Condition' may be a combination of different flags for specifying complex conditions like: 'ib's must be available from all input edges', 'from at least one input edge', 'from a user threads', etc.

Service routines. A service routine implements the actual functionality of the operation performed by a service. When this procedure accesses the data stored in its vertex, mutual exclusion is implicit for the procedure's non-preemptive execution. When the procedure is executed by a preemptible thread, it must use the synchronization primitives offered by the thread library to protect the vertex's private data. The address of the vertex' private data and the address of the parameter block referenced in the invocation block are generated using the macros:

```
top_data_p( ib )
top_param_p( ib )
```

Once completed, a service that wishes to send output parameters to other vertices, or to a user thread, can activate its postcondition procedure with the library routine:

```
void top_postcond( ib );
```

Postconditions. As with preconditions, all postconditions are executed non-preemptively. A postcondition defines a service's output policy. Specifically, each vertex contains an output queue for temporary storage of output ib's, and the DSA library offers the aforementioned access routines for queue manipulation:

```
void     top_enqueue_output( ib );
top_ib_t top_dequeue_output( service_id, tag );
bool     top_check_output( service_id, tag, condition );
```

A postcondition procedure can use these routines to define an output propagation policy for its vertex.

The most important action taken by postconditions is to generate vertex output. The following routines are used for output generation:

```
TOP_RESULT top_output_edges( ib );
TOP_RESULT top_output_vertex( ib, vertex_id );
TOP_RESULT top_output_user( ib );
```

'Top_output_edges' sends a copy of the specified invocation block across all of the vertex' output edges. For exception handling or when a vertex' output edges cannot be defined as part of the object's creation, the precondition procedure can alternatively use the routine 'top_output_vertex', which sends a copy of invocation block only to the single specified vertex. This routine is particularly useful when an object's communication structure is constructed dynamically, such as in dynamic broadcast trees, or for message routing in distributed systems. Finally, 'top_output_user' is used for transmission of results to a user thread. Such transmissions are performed via the vertex's output queue. Namely, the routine first enqueues the specified invocation block on the output queue and then checks if a user thread is waiting for it. If a thread is waiting, the routine signals it (i.e., puts the thread back in the processor's ready queue).