Paul Lu
University of Alberta

Implementing Scoped Behavior for Flexible Distributed Data Sharing

In the Aurora distributed shared data system, the programmer instantiates shared-data objects and uses scoped behavior to incrementally tune applications on a per-object and per-context basis. A class library implements shared-data objects as abstract data types and scoped behavior implements the optimizations within standard C++. Using a network of workstations connected by an ATM switch, the author demonstrates that Aurora performs comparably to message passing.

Parallel and distributed computing is a diverse area of research and practice.[1] Many hardware architectures have been proposed, but networks of workstations and clusters have recently garnered considerable attention. These distributed-memory platforms are attractive because of their ubiquitousness and good price performance, but they suffer from high communication overheads. Sharing data between distributed memories is more expensive than sharing data using hardware-based shared memory. Because an application's data-sharing policies determine how often, when, and what mechanisms are used for communications, they dramatically affect performance and must be optimized. (See the "Approaches to distributed data sharing" sidebar for a discussion of current approaches.)

The flexibility to tune a distributed-memory application varies depending on the kind of parallel programming system used. System architects can develop parallel applications using parallel languages (such as High Performance Fortran), libraries (Lapack, for example), runtime systems (such as Message Passing Interface), or a combination of these techniques. Each type of system has different strengths and weaknesses, but, generally speaking, high-level languages and shared-data systems are strong in ease of use, while message-passing systems are strong in performance. By design, high-level abstractions hide low-level details, such as when and what data is communicated. Conversely, message passing explicitly shows when data is sent and received. With sufficient (and often substantial) programming effort, a message-passing program can be highly tuned. Ideally, we would like to have both a high level of abstraction *and* the flexibility to tune a parallel application.

We have developed a novel technique, called *scoped behavior*, that provides a high degree of flexibility in applying an optimization within a high-level parallel programming system.[2] This article shows how application programmers use and implement scoped behavior and how three optimized applications perform on a network of workstations.

Flexibility using scoped behavior

In Aurora, the basic shared-data model is that of a distributed vector object or a distributed scalar object. Each object is an independent unit of sharing. The data encapsulated in the object can be accessed

1092-3063/00/$10.00 © 2000 IEEE

Approaches to distributed data sharing

Performance and usability are common but sometimes-conflicting design goals for parallel programming systems. On the one hand, low-level control of communication operations can give message passing a flexibility and performance advantage. The key mechanisms of message-passing systems are explicit message sends and receives for remote data. Local data is accessed using familiar reads and writes, as in sequential programs. Practically everything else about a message-passing program depends on how the programmer chooses to implement the application. The relative lack of constraints on message passing is the source of both its expressive power and programming complexity.

On the other hand, systems based on shared-memory and shared-data models are becoming increasingly popular for distributed applications. These systems provide an abstraction that allows local and remote data to be accessed using the same programming interface, such as loads and stores, or reads and writes. With a uniform interface, there is no need to mix local accesses with explicit message passing, making its use more convenient and less error-prone. Consequently, a variety of software-based logically shared systems for distributed-memory platforms have emerged. The systems emphasize usability, but they improve performance through a variety of optimizations. Broadly speaking, there are *distributed shared memory*[1,2] and *distributed shared data*[3,4] systems. Table A compares the characteristics of "typical" DSM and DSD systems.

Two sides

At one end of the shared-data spectrum, DSM systems use software to emulate hardware-based shared memory. Typically, DSM systems rely on fixed-sized units of sharing, often a page, because they use the same mechanisms as for demand-paged virtual memory. The virtual-memory space is partitioned into pages that hold private data and pages that hold shared data. Different processor nodes can cache copies of the shared data. As with hardware-based shared memory, a C-style pointer (such as `int *`) can refer to and name either local or remote data.

By manipulating the memory protection bits associated with a page, the DSM software can force a page fault into its own handler and selectively intervene when a page of shared data is accessed. The intervention is transparent to the programmer in the same way that virtual memory faults in the operating system are also transparent. But, instead of using a traditional backing store, the DSM runtime system communicates with other processor nodes to update, inval-

Table A. Distributed shared memory versus distributed shared data.

Factors	DSM	DSD
Key mechanisms	Page faults for virtual memory	Abstract Data Type and programmer's interface
Naming	Pointer	Object pointer or data descriptor
Unit of management	Fixed: page	Variable: object or region
Unit of sharing policy	All shared pages; sometimes per-page	Object or region
Can suffer from false sharing?	Yes	No
Can alter sharing policy per context?	Possible, but still problematic	Yes

from any processor node. In keeping with an abstract data type approach, shared-data objects are created, accessed, and destroyed using a programmer's interface. By exploiting various abstraction and object-oriented mechanisms in C++, it is possible to automate and hide the low-level details of the programmer's interface.

Specifically, C++ constructors and destructors hide resource allocation and deallocation details, such as memory for data buffers and caches. Also, the objects have internal data structures to keep track of the location and status of the shared data. Overloaded operators let the shared data be accessed using normal C++ syntax by translating a read or write access into an appropriate communication operation.

In concert with the basic shared-data objects, Aurora uses scoped behavior to provide the following:

- *Per-object flexibility*—the ability to apply an optimization to a specific shared-data object without affecting the behavior of other objects. Within a context, different objects can be optimized in different ways (heterogeneous optimizations).
- *Per-context flexibility*—the ability to apply an optimization to a specific portion of the source code. Different portions of the source code (such as different loops and phases) can be optimized in different ways.

I'll discuss the implementation details later, but first let's look at Aurora's higher-level programming abstractions.

A simple loop

Figure 1a demonstrates how to instantiate and access a distributed-vector object. `GVector` is a C++ class template provided by Aurora. Any built-in data type or user-defined concrete type[3] can serve as the template argument. The vector's size is a parameter to the constructor and, currently, the vector elements are block distributed across the processor nodes. Therefore, `vector1` is a vector object with 1,024 integer elements that are block distributed.

The programmer can assign values to the elements of `vector1` using the same syntax as with any C++ array. The overloaded subscript operator (`operator[]`) is an access function that determines whether the update to `vector1` at index `i` is to local or to remote data. If the data

idate, and otherwise manage the data in a shared page. Often, a single data-sharing policy serves for all shared pages, although some DSM systems support per-page policies.

At the other end of the spectrum, DSD systems treat shared data as an *abstract data type*. Instead of depending on page faults, a programmer's interface is used to detect and control access to the shared data. The access functions implement the data-sharing policy. If an object-oriented language is used, the ADT can be a shared-data object. Alternatively, the ADT can be implemented by a data descriptor and associated library functions. For example, the data descriptor can point to a contiguous region of memory that is designated for shared data.

A programmer's interface might not be as transparent as the virtual-memory approach, but there are some flexibility advantages to ADTs. For example, DSD systems can have a variable granularity of sharing because ADTs do not depend on the hardware's page size. Also, different data-sharing policies can be implemented as different ADTs. As the usage of the data structure changes, so will the data-access pattern of reads and writes, so the programmer can select the ADT with the most optimized policy for a given pattern. Consequently, the programmer can select the unit of sharing and the sharing policy to match the form and function of the particular data.

Value of flexibility

Flexibility in a shared-data system is important when dealing with the problem of false sharing and when optimizing data-access patterns in different computational phases.

False sharing occurs in page-based DSM systems when there is unnecessary communication between processes that do not actually share their data. Consider two processes that write to different portions of the same shared page. If the processes never read each other's updated values, there is no need for the processes to communicate. But, because data is managed and communicated on a per-page basis, the writes cause the entire page to be either updated or invalidated unnecessarily. Placing independent data on different pages can reduce false sharing, but this can lead to memory fragmentation. Furthermore, a shared page can exhibit false sharing in one computational phase and no false sharing in another phase. An inflexible unit of data management makes it difficult to eliminate false sharing under all circumstances.

DSD systems avoid false sharing by managing independent data as separate objects. And, different ADTs can implement different data-sharing policies. If one data structure is read-only and another is write-intensive, they can be optimized by different ADTs. Also, as part of the ADT approach, it is natural to consider different interfaces for the same shared data if the access pattern changes. If the output of one function serves as the input of another function, the same shared data might change from write-intensive to read-only. In a DSD system, changing a data-sharing policy can be as simple as changing the interface's access functions, without changing the encapsulated data.

To address the flexibility issue, the Aurora DSD system uses scoped behavior. As the programmer's interface for specifying a data-sharing optimization, scoped behavior allows each shared-data object and each portion of the source code (context) to be optimized independently of other objects and contexts.

References

1. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Conf. Principles and Practice of Parallel Programming*, ACM Press, New York, 1990, pp. 168–176.
2. C. Amza et al., "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer,* Vol. 29, No. 2, Feb. 1996, pp. 18–28.
3. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. Software Eng.*, Vol. 18, No. 3, Mar. 1992, pp. 190–205.
4. H.S. Sandhu, B. Gamsa, and S. Zhou, "The Shared Regions Approach to Software Cache Coherence," *Proc. Symp. Principles and Practices of Parallel Programming,* ACM Press, New York, 1993, pp. 229–238.

is local, a write is simply a store to local memory. If the data is remote, a write results in a network message. Similarly, a read access is either a load from local memory or a network message to get remote data. By default, the system reads from and writes to shared data synchronously, even if the data is on a remote node, because that data access behavior has the least error-prone semantics.

Now, for example, if a shared vector is updated in a loop *and* if the updates do not need to be performed immediately, we can optimize the loop by using release consistency.[4] Basically, all writes are buffered and updates to the vector will execute later, instead of synchronously. Three new elements are required to use scoped behavior to specify the optimization (see Figure 1b): opening and closing braces for the language scope and a system-provided macro. Of course, the new language scope is nested within the original scope and the new scope provides a convenient way to specify the context of the optimization.

The `NewBehavior` macro specifies that the release consistency optimization should be applied to `vector1`. Upon recompilation, and without any changes to the loop code itself, the behavior of the updates to `vector1` changes within the language scope. The new behavior uses buffers to batch the writes and auto-

```
GVector<int> vector1(1024);

for(int i = 0; i < 1024; i++)
  vector1[i] = someFunc(i);
```

(a)

```
GVector<int> vector1(1024);

{// Begin new language scope
  NewBehavior(vector1,GVReleaseC, int);

  for(int i = 0; i < 1024; i++)
    vector1[i] = someFunc(i);

}// End scope
```

(b)

Figure 1. Applying a data-sharing optimization using scoped behavior: (a) original loop; (b) optimized loop using scoped behavior.

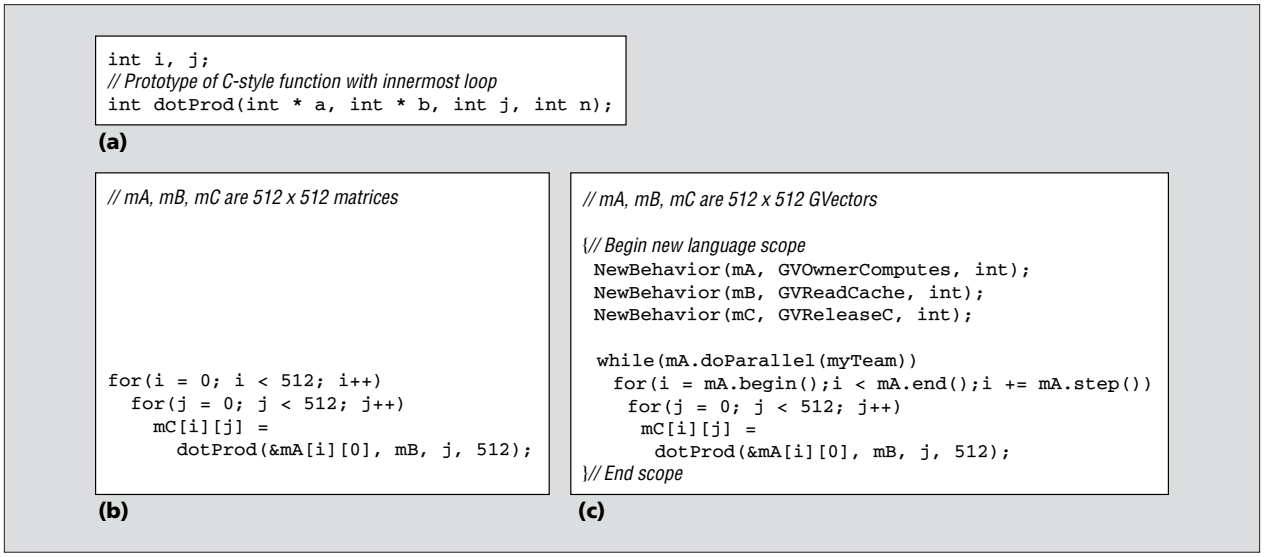

Figure 2. Matrix multiplication in Aurora: (a) common preamble, (b) sequential code, and (c) optimized parallel code.

Table 1. Some scoped behaviors.

Scoped behavior	Description
Owner-computes	Threads access only colocated data
Caching for reads	Create local copy of data
Release consistency	Buffer write accesses

matically flushes the buffers when the scope is exited.

Matrix multiplication

Let's now examine a more complex example of using scoped behavior. Consider the problem of nonblocked, dense matrix multiplication, as shown in Figure 2.

The basic data-parallel process model is that of teams of threads operating on shared data in single-program, multiple-data (SPMD) fashion. The preamble is common to both the sequential and parallel codes (Figure 2a). The basic algorithm consists of three nested loops, where the innermost loop computes a dot product and can be factored into a separate C-style function.

Conceptually, we can view an optimization as a change in the type of shared object for the lifetime of the scope. As an example of per-object flexibility, we apply three different data-sharing optimizations (Table 1) to the sequential code in Figure 2b to create the parallel code in Figure 2c. The first scoped behavior requires some modest change to the source code, but the last two behaviors require no changes:

1. `NewBehavior(mA,GVOwnerComputes,int)`: To partition the parallel work, the owner-computes technique is applied to distributed vector `mA`. Within the scope, mA is an object of type `GVOwnerComputes` and has special methods `doParallel()`, `begin()`, `end()`, and `step()`. Only the threads (each represented by a local myTeam pointer) that are colocated with a portion of mA's block-distributed data actually enter the `while`-loop and iterate over their local data. `dotProd()` expects pointers for parameters. Therefore, `GVOwnerComputes` provides a C-style pointer to the local data so that `dotProd()` executes with maximum performance. Although some changes to the source code are required to apply owner-computes, they are relatively straightforward.
2. `NewBehavior(mB, GVReadCache, int)`: To automatically create a local copy of distributed vector mB at the start of the scope, because it is read-only and reused many times, its type changes to `GVReadCache`. The scoped behavior of a read cache also allows dot`Prod()` to be called with C-style pointers that point to the cache. No lexical changes to the loop's source code are required for this optimization.
3. `NewBehavior(mC, GVReleaseC, int)`: To reduce the number of update messages to elements of distributed vector mC during the computation, its type changes to `GVReleaseC`. As with the simple loop example, the overloaded operators batch the updates into buffers, and messages only send when a buffer is full or when the scope is exited. Also, this optimization allows multiple writes to the same distributed vector and requires no lexical changes to the source code.

This heterogeneous set of optimizations lets the nested loops execute with far fewer remote data accesses than before. All read accesses are from a cache or local memory; all write accesses are buffered.

We can also exploit the high-level semantics of scoped behaviors to reduce communication overheads. For example, typical DSM systems send an individual request message for each page of remote data. Without any knowledge on the specific data-access pattern, the DSM system must use the most general-purpose policy, such as demand paging. However, scoped behaviors do contain extra semantic information. In particular, the read-cache scoped behavior specifies that *all* of vector `mB` in matrix multiplication is cached; therefore, there is no need to transfer each unit of data separately. We can eliminate the multiple request messages if the data is streamed into each read cache via bulk data transfer. Of course, bulk data transfer is not unique to Aurora, but scoped behavior's high-level seman-

tics and flexibility provide a natural conceptual and implementation framework.

Programming in Aurora

Aurora does not automatically parallelize an application. The typical methodology for developing and porting applications to Aurora "by hand" consists of three main steps:

- Shared arrays and shared scalars convert to `GVectors` and `GScalars`. Although the default synchronous access policy can be slow, we can optimize its performance after the program has been fully debugged.
- The parallel work is partitioned among the processors and threads. Owner-computes and SPMD-style parallelism are common and effective strategies, but the application programmer is free to implement other work-partitioning schemes as well.
- Various data-sharing optimizations can be tried on different bottlenecks in the program and on different shared-data objects. Often, the only required changes are a new language scope and a `NewBehavior` macro. Sometimes, straightforward changes to the looping parameters are needed, such as for owner-computes.

By limiting the number of required changes to the user's source code, scoped behavior makes it easier to experiment with different optimization strategies. For example, in the matrix multiplication program, we can apply owner-computes to vector `mC` instead, with read caches used for both vector `mA` and vector `mB`. The `dotProd()` function and the data-access source code remain unchanged. Reverting to the original strategy is also relatively easy. For the application programmer, the ability to experiment with different optimizations, with limited error-prone code changes, can be valuable.

Scoped behavior

Scoped behavior is a change in an ADT's interface for the lifetime of a language scope. For application programmers, scoped behavior is how they apply an optimization to a shared-data object. For system and class designers, scoped behavior is a collaboration between classes that changes the implementation of selected methods.

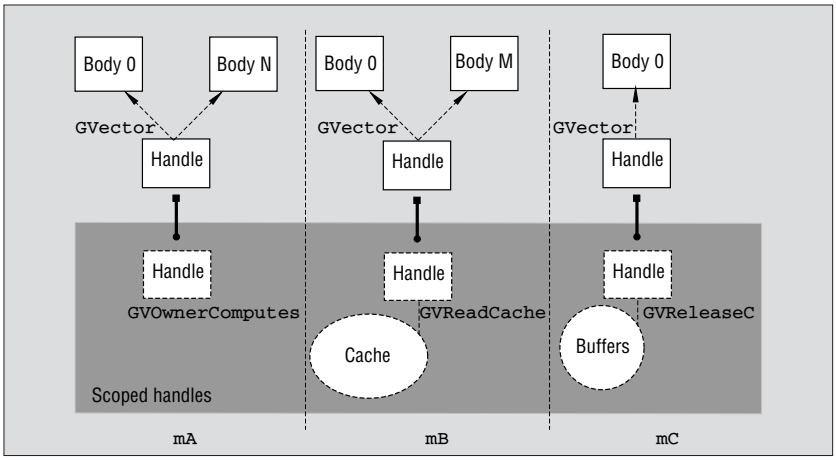

Figure 3. Composite objects in matrix multiplication.

Handle-body composite objects

Developers have explored some of the ideas behind scoped behavior as part of the handle-body and envelope-letter idioms in object-oriented programming,[3] the strategy design pattern,[5] and parametric shared regions in ABC++.[6] Scoped behavior builds upon these ideas.

Composite objects, such as handle-body objects, are multiple objects that behave as if they were a single entity.[3] In Aurora, the *handle object* defines the programmer's interface to the shared data and the *body object* (or objects) contain the actual data. Having multiple handles for the same body objects is a convenient way to support different ADT interfaces to the same encapsulated data. Depending on which handle is in use (in scope), the methods and behaviors will be different. I'll discuss the interaction between handle and body later, but first I'll focus on how different handles interact.

Language scopes and scoped handles

Language scopes are used to define the context of scoped behavior to exploit the compile-time property of name hiding and the runtime properties of object creation. Many block-structured languages can reuse an identifier within a nested language scope, thus hiding the identifier outside of the scope. A handle within a language scope that hides a handle outside of the scope is a *scoped handle*.

Figure 3 shows the relationship between the various collaborating objects inside the language scope of the matrix multiplication example. The solid boxes show the original `GVector` handle-body objects for `mA`, `mB`, and `mC`. In the example, each of the `GVectors` has a different number of body objects.

The dashed boxes (highlighted in gray) show the scoped handles used to implement the owner-computes, caching for reads, and release consistency behaviors. Inside the scope, the `GVector` handles are hidden and unused, but the scoped handles can access the data in the body objects via a reference (for example, a pointer) to the original handles. Dynamic actions can be associated with the construction and destruction of the scoped handles, such as creating, flushing, and destroying cache and buffer objects.

Handle references and alternate interfaces

As Figure 4a shows, Aurora provides the scoped behavior macro `NewBehavior` to help establish the reference from one handle to another. Figure 4b shows the original programmer's source code and Figure 4c shows the code after the standard preprocessor of the C++ compiler has expanded the macro.

The `NewBehavior` macro is parameterized by the name of the original shared-data object (`ORIG`), the scoped behavior or type of the new scoped handle (`SB`), and the type of the vector elements (`TYPE`). (This is a multiline macro and the `##` symbol is the standard preprocessor operator for lexical concatenation. Also, the prefix `AU_` is arbitrary and can be redefined, if necessary. Unfortunately, the more concise syntax of `GVReleaseC<int> vector1(vector1)` conflicts with the C++ standard's semantics. According to the standard, the new `vector1` is passed a reference to

```
#define NewBehavior(ORIG, SB, TYPE) \                    // Macro provided by aurora.H
  GPortal< GVector<TYPE> > AU_ ## ORIG(ORIG); \
  SB< TYPE > ORIG(AU_ ## ORIG);

template <class C_OrigHandle>                            // Class template provided by aurora.H
class GPortal
{
  private:
    C_OrigHandle * save;                                 // Saved handle
  public:
    GPortal(C_OrigHandle & h) { save = &h;}              // In: Constructor
    operator C_OrigHandle &() { return *save;}           // Out: Type conversion operator
};// GPortal
```

(a)

```
GVector<int> vector1(1024);

{// Begin new language scope
 NewBehavior(vector1, GVReleaseC, int);


  for(int i = 0; i < 1024; i++)
    vector1[i] = someFunc(i);
}// End scope
vector1[0] = 1;                  // Synchronous update
```

(b)

```
GVector<int> vector1(1024);

{// Begin new language scope
  GPortal<GVector<int> > AU_vector1(vector1);
  GVReleaseC<int> vector1(AU_vector1);

  for(int i = 0; i < 1024; i++)
    vector1[i] = someFunc(i);
}// End scope
vector1[0] = 1;                  // Synchronous update (still)
```

(c)

Figure 4. Aurora's scoped behavior macro: (a) the macro, (b) source code, and (c) code after standard preprocessor pass.

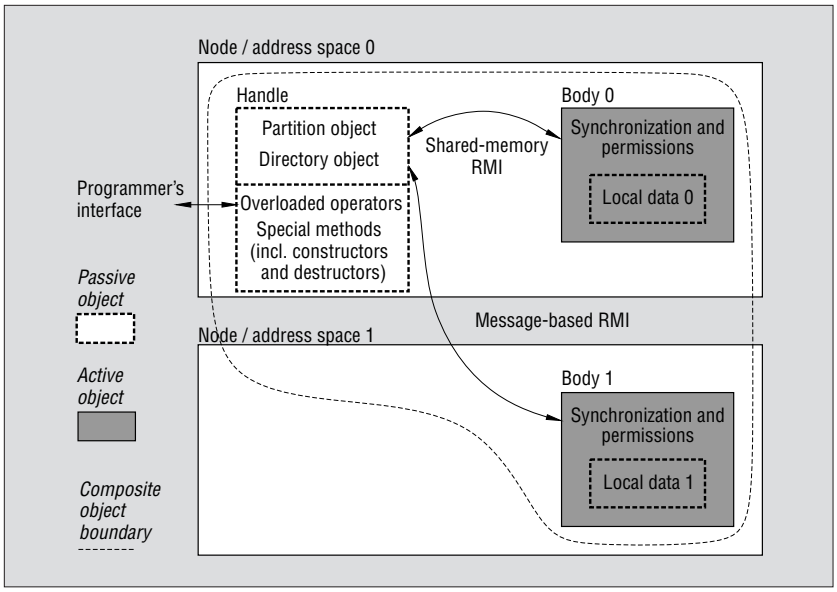

Figure 5. Shared-data composite objects.

itself, instead of to the original object, thus requiring an intermediary `GPortal` object. Fortunately, the macro hides the existence of the intermediary object.)

The macro actually instantiates two objects. The first object, `AU_vector1`, is of type `GPortal`. Its sole function is to save a pointer to the original `vector1` object. The second object, the new scoped handle `vector1` of type `GVReleaseC<int>`, hides the original object but can access its internal state using the pointer passed by `AU_vector1`. Thus, the scoped handle can delegate, mimic, or change the original shared-data object's functionality, and the user's source code does not change. In other words, the programmer's ADT interface changes without the encapsulated data or the user's source code changing.

Because the scoped handle has the same name as the original `vector1`, the compiler will generate the loop body code according to class `GVReleaseC` instead of the original object's class. The class template `GVReleaseC` behaves exactly like `GVector`, except that the overloaded operators now buffer the updates and the destructor flushes the buffers at the end of the scope. Again, we can conceptualize scoped behavior as using the `NewBehavior` macro to temporarily change the type of the original object.

The source code outside of the context of the optimization continues to refer to the original `GVector`. Therefore, synchronous updates remain the default behavior outside of the scope, illustrating per-context flexibility.

Shared-data class library

Let's now take a detailed look at the design and implementation of the C++ classes for the shared-data objects and data-sharing optimizations. By design, these classes collaborate to support scoped behavior.

Shared-data composite objects

As discussed, the class library uses the handle-body idiom to create composite objects for shared data (Figure 5). In addition to simplifying the implementation of scoped behavior, the extra level of indirection between handle and body allows for

1. *Data distribution*. A distributed vec-

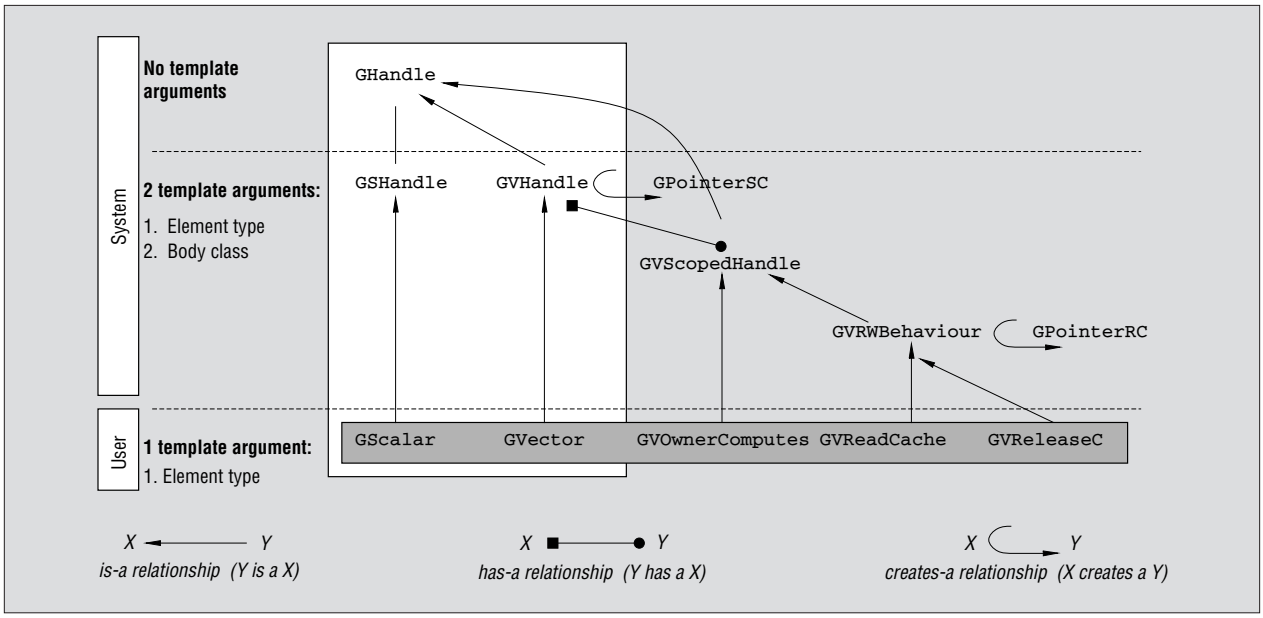

Figure 6. Class hierarchy for handles.

tor is a set of body objects, and each body object can reside in a different address space or on a different processor node. The handle includes a *partition object* to abstract the distribution strategy and a *directory object* to keep track of the location of the bodies. A distributed scalar has a single body object. Figure 5 shows a distributed vector object with a handle and two body objects, where one of the body objects is on a different node than the handle.

2. *Location-transparent data accesses.* Through overloaded operators in the handle, programmers can access the distributed data through a uniform interface, regardless of the location of the actual data. Thus, for a given vector index, the partition object determines which body holds the data and the directory object provides a pointer to the body object.
3. *Cheaper parameter passing of shared data.* Only handles pass across function calls; the data in the bodies is not copied. Handles can also pass between address spaces, if desired, because the partition and directory objects are sufficient to locate any body object from any address space.

For performance-sensitive functions, such as `dotProd()` in Figure 2, the handle-body indirection's overheads can be avoided in controlled ways through *type conversion operators* that return C-style pointers. (Actually, the proper name is simply *conversion operator*. We are more verbose to be more descriptive.) In C++, a type conversion operator is a method that converts an object of one type to an object of a different, but "compatible," data type. Type conversion operators are a more powerful and flexible form of type casting. Both class `GVOwnerComputes` and class `GVReadCache` define a type conversion operator that, because `dotProd()` is expecting pointers as parameters, converts between a handle object and a pointer to the same data.

Aurora's current implementation creates handles as regular C++ objects. However, it implements each individual body as an active object, an object with its own thread of control, which is useful for implementing any necessary synchronization behavior. The body classes support `get()` and `put()` data-access methods, including batch update and block-read variations. Handle and body interact using the remote method invocation (RMI) mechanism provided by ABC++.[6] The runtime system automatically selects between shared-memory and message-based communication (that is, MPI) mechanisms for transmitting RMIs.

Class hierarchy for handles

Because most of the data-sharing functionality is implemented in the handles, this discussion will focus on the handle classes. Figure 6 is a diagram of the main classes in the hierarchy of shared-data handles. (The notation is based on that of Grady Booch but with some simplifications and changes to better suit this presentation.) In general, the application programmer need only use the classes at the leaves of the hierarchy (labeled "user" and highlighted in gray). These classes hide the more complex templating and class hierarchy considerations with which the "system" must deal.

The *is-a* relationship in Figure 6 is the usual notion of inheritance. Class Y is a subclass of X so "Y is a X." For example, `GVHandle` (`V` is for vector, of course) is a subclass of `GHandle`, so an object of class `GVHandle` is also of class `GHandle` (see also Figure 7). In fact, class `GHandle` is the base class for all handles. Common access methods are factored into the base class.

The *has-a* relationship exists when an object contains a reference or pointer to an instance of another class. If "Y has a X," an object of class Y contains a reference or pointer to an object of class X. With the right access control permissions, Y can

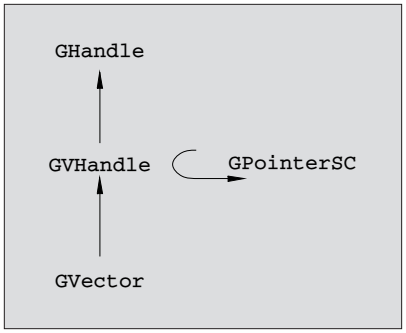

Figure 7. Class hierarchy for `GVector`.

Table 2. Object model for handles.

		Data-Sharing Optimizations		
Object	**GVector**	**GVOwnerComputes**	**GVReadCache**	**GVReleaseC**
Internal State	Partition and directory objects	Handle reference, pointer to local data	Handle reference, cache object	Handle reference, buffer objects
Enter scope (Constructor)	Create body objects	Save reference, find local data	Save reference, create and load cache	Save reference, allow buffer creation on demand
Within scope	Synchronous reads and writes using `GPointerSC`	Access local data using pointer	Read from cache, synchronous updates using `GPointerRC`	Synchronous reads, buffer updates using `GPointerRC`
Exit scope (Destructor)	Free body objects	No action needed	Free cache	Flush and free buffers

access the internal state of X. Class Y can also call any method in class X to delegate functionality and behavior. In general, the has-a relationship includes the case where Y contains an instance of X. However, in these classes from Aurora, the has-a relationship is implemented using a pointer and not an instance.

The *creates-a* relationship exists when at least one of a class's methods returns an object of another class. If "X creates a Y," then an object of class X creates and returns an object of class Y in one of its methods. For example, an overloaded subscript operator (that is, `operator[]`) can return a temporary object that encodes information about a specific vector element.

The C++ object model provides a convenient create-use-destroy framework within which to implement composite objects (Table 2). For the basic shared-data vector, the relevant classes in the hierarchy are `GHandle`, `GVHandle`, and `GVector` (Figure 7). When the application programmer instantiates a `GVector`, the class constructor transparently creates the body objects. Later on, the destructor automatically frees the body objects. During the object's lifetime (that is, within scope), class `GVHandle` contains the partition and directory objects needed to locate and communicate with the body objects. To read and write data, the overloaded subscript operator of `GVHandle` returns an object of type `GPointerSC`, which is a *pointer object*. When evaluating C++ expressions involving objects and overloaded operators, temporary objects represent the result of subexpressions.[3] Reading from or writing to the vector element invokes the appropriate method (type conversion operator) and the overloaded assignment operator of `GPointerSC`, resulting in a synchronous remote memory access. Further details and sample C++ code are available elsewhere.[2]

Data-sharing optimizations: scoped handles

We combined the create-use-destroy object model with handle references to implement the scoped handles (Table 2). For the data-sharing optimizations, the parent class `GVScopedHandle` extracts and maintains a reference to a `GVHandle`, as per the has-a relationship (Figures 6 and 8). The partition and directory objects of the `GVHandle` are not copied, thus minimizing the construction costs of a scoped handle.

```
// Template argument C_Data is the element type; C_LV is the body class.
template <class C_Data, class C_LV>
class GVScopedHandle : public GHandle                          //is-a GHandle
{
  protected:
      // I am a "friend" of GVHandle.
    GVHandle<C_Data, C_LV> * origHandle; // To access internal state of original object (has-a)
    //...other data members...
  public:
    GVScopedHandle(GVHandle<C_Data, C_LV> & gv)   // Construct with original handle
        { origHandle = &gv;}                                   // Cache the handle
    ~GVScopedHandle();
    //...other methods...
}; // GVScopedHandle (System)
```

Figure 8. Handle references: `GVScopedHandle`.

Class `GVOwnerComputes`, in its constructor, uses the handle reference to determine the address of the local (co-located) body object's data. Therefore, `GVOwnerComputes` can return a C-style pointer from the appropriate type conversion operator and from the overloaded subscript operator. Then, the local data is accessed using the pointer. As I've discussed, `GVOwnerComputes` also defines special functions to support easy iterating over the local data.

We have implemented most of the read-write functionality of the caching for reads and release consistency behaviors within a cache and buffer manager class. Class `GVRWBehavior` can optionally create a read cache for shared data and create update buffers to shared data (Figure 9). Classes that derive from `GVRWBehavior` explicitly configure the caching and buffering options. The overloaded subscript operator in `GVRWBehavior` returns an object of class `GPointerRC`, which is similar in concept to class `GPointerSC`, but with two important differences. First, if the read cache exists and is loaded, `GPointerRC` is configured to access data from the cache instead of from the remote body. Second, if the update buffers are enabled, `GPointerRC` is configured to store updates in the buffer rather than initiate a remote memory access. The buffers are created on demand. Depending on the configuration of the cache and buffers, `GPointerRC` will access the shared data appropriately.

Therefore, the constructor of class `GVReadCache` calls the appropriate `GVRWBehavior` methods to create and

```
// Template argument C_Data is the element type; C_LV is the body class.
//  Classes Cache, BatchWrite, and GPointerRC are provided by Aurora.
template <class C_Data, class C_LV>
class GVRWBehavior : public GVScopedHandle<C_Data, C_LV>                    //is-a GVScopedHandle
{
 protected:
   Cache<C_Data, C_LV> * readCache;                                          //Configurable read cache
   BatchWrite<C_Data, C_LV> * updateBuf[MAX_LOCALS];      //Configurable buffers for release consistency
     //...other data members...
 public:
   GVRWBehavior(GVHandle<C_Data, C_LV> & gv) :                        //Construct with original handle
           GVScopedHandle<C_Data, C_LV>(gv) {}
   ~GVRWBehavior();                                        //Destructor flushes update buffers if necessary
   createCache();                                                        //Method to create read cache
   allowUpdateBuf();                                                    //Method to allow update buffers
   GPointerRC<C_LV, C_Data> operator[] (int index);          //Pointer object to cache/buffer (creates-a)
   //...other methods...
};//GVRWBehavior (System)

// Template argument C_Data is the element type.
// LVector (provided by Aurora) is the body class.
template <class C_Data>
class GVReleaseC : public GVRWBehavior<C_Data, LVector<C_Data> >          // is-a GVRWBehavior
{
 public:
   GVReleaseC(GVector<C_Data> & gv) :                     //Original handle via GPortal/NewBehavior macro
     GVRWBehavior<C_Data, LVector<C_Data> >(gv)
     { allowUpdateBuf();}                                          //Construct to allow update buffers
   ~GVReleaseC();
 //...inherits operator[] and other methods...
}; // GVReleaseC (User)
```

Figure 9. Interface for release consistency scoped behavior: `GVReleaseC`.

load the read cache. Similarly, the constructor of class `GVReleaseC` calls the appropriate `GVRWBehavior` method to enable the use of update buffers. The destructor for class `GVRWBehavior` makes sure all buffers are flushed.

Performance evaluation

Let's now examine the performance and data-sharing overheads of applications implemented using both Aurora and a MPI-based message-passing system. We found that Aurora performed comparably to message passing for the three applications considered.

The hardware platform used for these experiments is a cluster of PowerPC 604 workstations with 133-MHz CPUs, 96 Mbytes of main memory, and a 155 Mbps ATM network with a single switch. The software includes IBM's AIX 4.1 operating system, Posix threads, the ABC++ class library, and the MPICH (version 1.1.10) implementation of MPI. MPICH serves as part of the runtime system for ABC++ (and thus Aurora, too) and as the baseline message-passing system. For our platform, MPICH uses sockets and TCP/IP. Of course, there are multiple implementations of the MPI standard, each with their performance strengths and weaknesses. Therefore, for precision, I will refer to our message-passing programs as MPICH programs for the rest of this discussion.

The applications are a matrix multiplication program (using 704×704 matrices), a 2D diffusion simulation (a 1536×1536 grid is simulated for 32 timesteps), and a parallel sort (using 8 million random integer keys) via the Parallel Sorting by Regular Sampling (PSRS) algorithm.[7] The Aurora and MPICH implementations share much of the source code's sequential portions. Figure 10 shows the respective speedups for up to eight processors. Speedups are computed against C implementations of the same algorithm (or against quicksort in the case of the parallel sort). Overall, the Aurora and MPICH programs have very similar speedups, which is encouraging for Aurora because message-passing programs are generally acknowledged to set a high standard of performance.

The matrix multiplication program consists of two phases separated by a barrier. The same matrix multiplication function, with scoped behaviors, serves for both phases, but the function is called with different shared-data objects as parameters. Phase 1 computes $P \rightarrow Q \times R$. Phase 2 computes $R \leftarrow Q \times P$. In contrast to Figure 2, the owner-computes scoped behavior is applied to both `mA` and `mC` and the read cache scoped behavior is applied to `mB`. This makes it easier to evaluate the read cache's overheads because the owner-computes optimization does not involve any communication overhead. Although the specific matrix computation is synthetic, we designed it to demonstrate how different scoped behaviors can be applied to a shared-data object in different computational phases.

For matrix multiplication, Aurora achieves higher speedups than MPICH, especially for eight processors, due to the optimizations provided by the read cache scoped behavior. We isolated and measured the data-sharing overheads (Figure 11). As previously discussed, we can

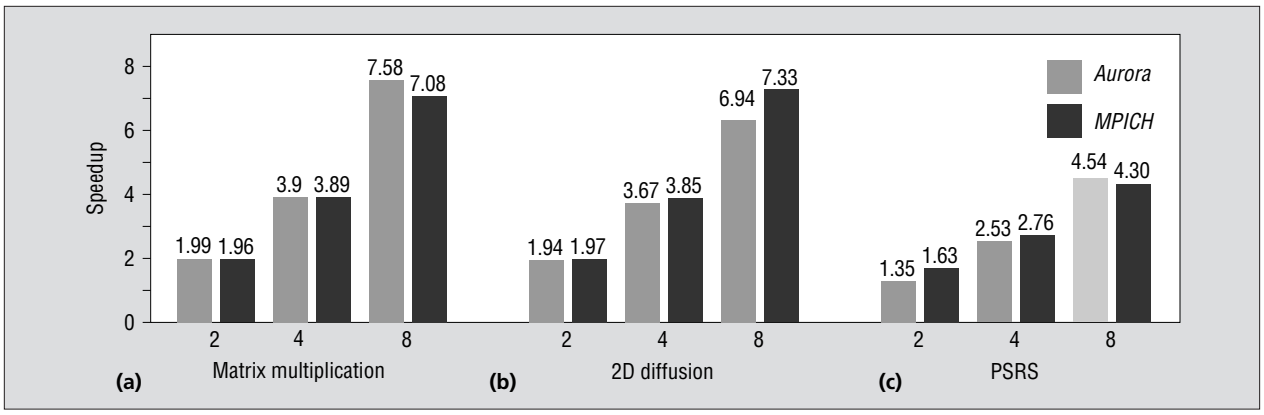

Figure 10. Speedups for Aurora and MPICH Applications: two, four, and eight processors, 155 Mbps ATM: (a) matrix multiplication, (b) 2D diffusion, and (c) PSRS.

exploit the high-level semantics of using read caches by using bulk data-transfer protocols to fill the caches. Aurora's real-time overheads are between 16% and 56% less than MPICH for this particular data-sharing pattern. Aurora outperforms MPICH in large part because Aurora uses UDP/IP for bulk data to avoid some of the protocol overheads associated with TCP/IP.

By using UDP, Aurora bypasses TCP's congestion avoidance algorithms and flow-control mechanisms. In an all-to-all data-sharing pattern, there will be congestion and contention. But, whereas TCP will conservatively back off before retransmitting to avoid flooding a shared network, a UDP-based approach can retransmit immediately under the assumption that the network is dedicated to the task at hand. If a network is not shared, waiting too long before retransmission wastes network bandwidth. The performance advantage of Aurora versus MPICH increases with the number of processors. As the number of processors increases, the opportunities for congestion (*n* processors sending to one processor, for example) also increases, which indicates a scalability issue with TCP (or our system's implementation of TCP) for this communication pattern. TCP's robust and conservative approach is well suited for shared wide-area networks, but it is not optimal for dedicated local-area networks and this type of data sharing.

Strictly speaking, MPICH could be modified (in the future) to use UDP and support a bulk data transfer protocol. In that case, the read cache scoped behavior would simply use the new functionality. Scoped behavior is meant to be a high-level abstraction and framework to exploit optimization mechanisms in the lower layers of the software, whether it is UDP or MPICH.

The 2D diffusion application simulates the diffusion of matter over time by computing a nine-point stencil function at all points on a grid. Most of the computation involves only accesses to local data, but there is data sharing between processes at the borders of the block-distributed vector that represents the grid. The MPICH version of 2D diffusion achieves somewhat higher speedups than the Aurora version because, with a message-passing approach, there is no need to explicitly synchronize at the end of a time step because the exchange of data can be an implicit synchronization. A shared-data approach typically requires an explicit and separate barrier, with associated overheads, to prevent the premature transfer of data.

PSRS is a multiphase parallel sorting algorithm that includes a communication-intensive data exchange phase for keys,[8] which limits the speedups. The MPICH version of PSRS achieves higher speedups than the Aurora version for the two- and four-processor cases. In the PSRS algorithm's data-exchange phase, the amount of data to be communicated can vary depending on the specific input data.[7] This is in contrast to the "always exchange all the data" semantics of a read cache in matrix multiplication. So, PSRS cannot use the previous scoped behavior and does not yet implement a new data-sharing optimization.

Interestingly, in the eight-processor case, Aurora outperforms MPICH. Although there is no specific scoped behavior to optimize this data exchange, the flow-control strategy used in Aurora is more efficient than in MPICH, especially as the number of processors increases. Again, in fairness to MPI and MPICH, these overheads might be lower (or just different) with other implementations of MPI or on other hardware platforms. These comparisons are mainly intended to show that Aurora programs can approach the high overall performance of message passing. We're currently conducting a more detailed performance evaluation.

Discussion and related work

One disadvantage of the scoped behavior approach is that each different behavior requires additional implementation effort. Of course, it is the system designer who must implement the new scoped behaviors, not the application programmer. Fortunately, data-sharing patterns do reappear in different contexts[8] and we've found that scoped behaviors are highly reusable. If the experience with group operations in MPI is any guide, a small set of optimizations (with simple variations) can cover many of the interesting sharing patterns in real applications. Furthermore, although the current set of scoped behaviors is small, designers can combine the behaviors on a per-context and per-object basis to support a variety of optimization strategies. And, it is the per-context flexibility of scoped behavior that distinguishes Aurora from other systems.

There is already a large body of work in the area of DSM and DSD systems (see the "Approaches to distributed data sharing" sidebar). Related work in High Performance Fortran and parallel array classes has also addressed the basic problem of transparently sharing data.

We can optimize different access pat-

terns on shared data through type-specific protocols and runtime annotations. For example, Blizzard[9] and Munin[10] provide protocols customized to specific data-sharing behaviors. Runtime libraries, such as shared regions, associate coherence actions with access annotations (function calls). Unlike Munin, Aurora does not require special compiler support and different optimizations can be used in different contexts. Unlike Blizzard, Aurora integrates the optimizations into the programming language to generate custom code for different coherence actions, for added implementation and performance flexibility. Unlike function libraries, the automatic construction and destruction of scoped handles make it impossible for the programmer to omit an annotation and miss a coherence action.

Aurora's handle-body object architecture and the association of data movement with constructors and destructors are inspired by ABC++'s parametric shared region mechanism.[6] However, there are two significant differences. First, Aurora lets distributed vectors be partitioned between different address spaces to improve scalability and support owner-computes using multiple nodes. A parametric shared region in ABC++ has a single home node, so shared data cannot be partitioned. Second, Aurora supports multiple writers to the same distributed vector object, which can be important for performance,[11] while parametric shared regions only allow a single writer.

Notably, both ABC++ and scoped behavior share an important safety benefit with respect to exception handling. If a C++ exception is thrown within the scope, the class destructor of the scoped object has a opportunity to free resources, cleanup state information, and otherwise recover from the exception. Even in a sequential language, exception handling is a complicated issue. Parallel programming systems based on simple function libraries, language extensions, or custom compilers might not be able to provide the exception-handling functionality of C++ without substantial engineering effort. Fortunately, by staying within standard C++ and by exploiting language scopes, Aurora (and ABC++) gains all the existing exception handling infrastructure with no extra effort.

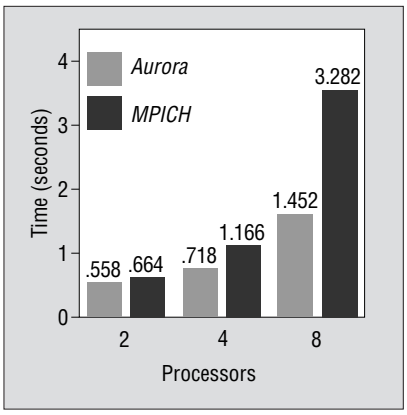

Figure 11. Data-sharing overheads in matrix multiplication: two, four, and eight processors.

WHEN DEVELOPING APPLICATIONS for distributed-memory platforms, such as a network of workstations, ease of use often makes shared-data systems preferable. Therefore, researchers have experimented with a number of DSM and DSD systems. However, page-based data management and inflexible sharing policies can result in unnecessary communication overheads and can make it more difficult to optimize some data-sharing patterns. A system that provides the benefits of a shared-data model and that can achieve performance comparable with a message-passing model is desirable.

The Aurora DSD system takes an abstract data type approach to a shared-data model. Given our current system's encouraging performance, we are exploring new scoped behaviors and developing more applications using Aurora.

Acknowledgments

I wish to thank Ben Gamsa, Eric Parsons, Karen Reid, Jonathan Schaeffer, Ken Sevcik, Michael Stumm, Duane Szafron, Greg Wilson, Songnian Zhou, and the anonymous referees for their comments and support during this work. This work was part of my PhD at the University of Toronto. Thank you to Toronto's Department of Computer Science and NSERC for financial support. Thanks also to ITRC and IBM for their support of the POW Project.

References

1. G.V. Wilson, *Practical Parallel Programming*, MIT Press, Cambridge, Mass., 1995.
2. P. Lu, "Implementing Optimized Distributed Data Sharing Using Scoped Behavior and a Class Library," *Proc. Third Conf. Object-Oriented Technologies and Systems (COOTS)*, Usenix, Berkeley, Calif., 1997, pp. 145–158; www.cs.ualberta.ca/~paullu.
3. J.O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, Reading, Mass., 1992.
4. S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer,* Vol. 29, No. 12, Dec. 1996, pp. 66–76.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, Mass., 1995.
6. W.G. O'Farrell et al., *Parallel Programming Using C++*, G.V. Wilson and P. Lu, eds., MIT Press, Cambridge, Mass., 1996.
7. X. Li et al., "On the Versatility of Parallel Sorting by Regular Sampling," *Parallel Computing*, Vol., 1993, pp. 1079–1103.
8. P. Lu, "Using Scoped Behavior to Optimize Data Sharing Idioms," *High Performance Cluster Computing: Programming and Applications*, Vol. 2, R. Buyya, ed., Prentice Hall, New York, 1999.
9. B. Falsafi et al., "Application-Specific Protocols for User-Level Shared Memory," Proc. Supercomputing, I*EEE Computer Soc. Press*, Los Alamitos, Calif., 1994, pp. 380–389.
10. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Conf. Principles and Practice of Parallel Programming,* ACM Press, New York, 1990, pp. 168–176.
11. C. Amza et al., "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer,* Vol. 29, No. 2, Feb. 1996, pp. 18–28.

Paul Lu is an assistant professor of computing science at the University of Alberta. His research interests span many aspects of high-performance computing and systems software, including parallel and distributed systems, and operating systems. As an MSc student at Alberta, he worked on the Chinook checkers playing program. His PhD is from the University of Toronto. He is a member of the IEEE and ACM. Contact him at the Dept. of Computing Science, Univ. of Alberta, Edmonton, Alberta, T6G 2H1, Canada; paullu@cs.ualberta.ca.