The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library

Paul Lu
Dept. of Computer Science
University of Toronto
Toronto, Ontario

Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library

Paul Lu
Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4
Canada
paullu@sys.utoronto.ca

Abstract

Sometimes, it is desirable to alter or optimize the behaviour of an object according to the needs of a specific portion of the source code (i.e., context), such as a particular loop or phase. One technique to support this form of optimization flexibility is a novel approach called *scoped behaviour*. Scoped behaviour allows the programmer to incrementally tune applications on a per-object and per-context basis within standard C++.

We explore the use of scoped behaviour in the implementation of the Aurora distributed shared data (DSD) system. In Aurora, the programmer uses scoped behaviour as the interface to various data sharing optimizations. We detail how a class library implements the basic data sharing functionality and how scoped behaviour coordinates the compile-time and run-time interaction between classes to implement the optimizations. We also explore how the library can be expanded with new classes and new optimization behaviours.

The good performance of Aurora suggests that using scoped behaviour and a class library is a viable approach for supporting this form of optimization flexibility.

1 Introduction

Optimizing a program's data access behaviour can significantly improve performance. Ideally, the programming system should allow each object to be optimized independently of other objects and each portion of the source code (i.e., context) to be optimized independently of other contexts. Towards that end, researchers have explored various compiler and run-time techniques to provide per-object and per-context flexibility in applying an optimization.

We describe how *scoped behaviour*, a change in the implementation of methods for the lifetime of a language scope, can provide the desired optimization flexibility within standard C++. A language scope (i.e., nested braces in C++) around source code selects the context and the re-defined methods implement the optimization. Scoped behaviour requires less engineering effort to implement than compiler extensions and it is better integrated with the language, thus less error-prone to use, than typical run-time libraries.

Specifically, we focus on a single application of scoped behaviour: supporting optimized distributed data sharing. Since this discussion is closely tied to a particular problem domain, we begin with a brief introduction to distributed data sharing. Then we provide an overview of the Aurora distributed shared data system [Lu97], detail how scoped behaviour and the class library are implemented, and discuss some performance issues.

2 Distributed Data Sharing

Parallel programming systems based on shared memory and shared data models are becoming increasingly popular and widespread. Accessing local and remote data using the same programming interface (e.g., reads and writes) is often more convenient than mixing local accesses with explicit message passing.

On distributed-memory platforms, the lack of hardware support to directly access remote memories has prompted a variety of software-based, logically-shared systems. Broadly speaking, there are *distributed shared memory* (DSM) [Li88, BCZ90, ACD+96] and *distributed shared data* (DSD) [BKT92, SGZ93, JKW95] systems. Support for distributed data sharing, whether it is page-based as with DSM, or object-based (or region-based) as with DSD, is an active area of research. The spectrum of implementation techniques spans special hardware support, run-time function libraries, and special compilers.

Layer	Main Components and Functionality
Programmer's Interface	Teams of threads for SPMD-style parallelism, active objects Distributed vector and scalar objects *Scoped behaviour*
Shared-Data Class Library	Handle-body shared-data objects Overloaded operators and special methods; immediate data access (default behaviour) Data sharing optimizations Owner-computes, caching data for reads, release consistency for writes
Run-Time System	Active objects and remote method invocation (currently, ABC++) Threads (currently, pthreads) Communication mechanisms (currently, shared memory and MPI)

Table 1. Layered View of Aurora

(a) Original Loop

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




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

(b) Optimized Loop Using Scoped Behaviour

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

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

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

Figure 1. Applying a Data Sharing Optimization Using Scoped Behaviour

In this context, the all-software Aurora DSD system provides a shared-data programming model on distributed-memory hardware. All shared data are encapsulated as objects and are accessed using methods. To overcome the latency and bandwidth performance problems of typical distributed-memory platforms, Aurora provides a set of well-known data sharing optimizations.

Although other DSM and DSD systems also offer data sharing optimizations, Aurora is unique in how these optimizations are integrated into the programming language. Pragmatically, scoped behaviour allows the applications to be incrementally tuned with reduced programmer effort. Also, as an experimental platform, Aurora's class library approach is relatively easy to extend with new behaviours. In particular, one of the goals of this research is to support common data sharing idioms, specified and optimized using scoped behaviour, with good performance.

3 Overview of Aurora

Aurora can be viewed as a layered system (Table 1). The key layers will be discussed later on, but we begin with a quick overview.

Application programmers are primarily concerned with the upper two layers of the system: the programmer's interface and the shared-data class library. The basic data-parallel process model is that of teams of threads operating on shared data in SPMD-fashion (single program, multiple data). The basic shared-data model is that of a distributed vector object or a distributed scalar object. Once created, a shared-data object is transparently accessed, regardless of the physical location of the data, using normal C++ syntax. By default, shared data is read from and written to immediately (i.e., synchronously), even if the data is on a remote node, since that data access behaviour has the least error-prone semantics.

Figure 1(a) demonstrates how a distributed vector object is instantiated and accessed. `GVector` is a C++ class template provided by Aurora. Any built-in data type or user-defined structure or class can be used as the template argument. The size of the vector is a parameter to the constructor and, currently, the vector elements are block distributed across the physical nodes.

Now, for example, if a shared vector is updated in a loop *and* if the updates do not need to be performed immediately, then the loop can use release consistency [GLL+90, AG96] and batch the writes (see Figure 1(b), shown side-by-side for easy comparison). Without any changes to the loop code itself, the *behaviour* of the updates to `vector1` is changed within the language scope.

(a) Common Preamble

```
int i, j;
// Prototype of C-style function with innermost loop
int dotProd( int * a, int * b, int j, int n );
```

(b) Sequential Code

```
// mA, mB, mC are 512 × 512 matrices

for( i = 0; i < 512; i++ )
  for( j = 0; j < 512; j++ )
    mC[i][j] =
      dotProd( &mA[i][0], mB, j, 512 );
```

(c) Optimized Parallel Code

```
// mA, mB, mC are 512 × 512 GVectors

{ // Begin new language scope
 NewBehaviour( mA, GVOwnerComputes, int );
 NewBehaviour( mB, GVReadCache, int );
 NewBehaviour( mC, GVReleaseC, int );

 while( mA.doParallel( myTeam ) )
   for( i = mA.begin();i < mA.end();i += mA.step())
     for( j = 0; j < 512; j++ )
       mC[i][j] =
         dotProd( &mA[i][0], mB, j, 512 );
} // End scope
```

Figure 2. Matrix Multiplication in Aurora

The `NewBehaviour` macro specifies that the release consistency optimization should be *applied* to `vector1`.

Therefore, scoped behaviour is the main interface between the programming model and the data sharing optimizations, providing:

- *Per-object* flexibility: The ability to apply an optimization to a specific shared-data object without affecting the behaviour of other objects. Within a context, different objects can be optimized in different ways (i.e., 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 (e.g., different loops and phases) can be optimized in different ways.

The lowest layer of Aurora, the run-time system, provides the basic thread management and communication mechanisms. The current implementation of Aurora uses the ABC++ class library for its *active object* mechanism, an object that has a thread of control associated with it, and *remote method invocation* (RMI) facilities [OEPW96]. RMIs are syntactically similar to normal method invocations, but RMIs can be between objects in different address spaces. If desired, the application programmer can directly utilize the active object and RMI mechanisms to implement a more control-parallel process model. Also, although ABC++ already has a parametric shared region (PSR) mechanism, it is not used by Aurora.

In turn, ABC++ uses standard pthreads [Pth94] for concurrency and either shared memory or MPI message passing [GLS94] for communication.

4 Programmer's Interface

A more detailed description of the programmer's interface to Aurora can be found elsewhere [Lu97], but we briefly touch upon the main ideas with an example.

4.1 Example: Matrix Multiplication

For illustrative purposes, consider the problem of non-blocked, dense matrix multiplication, as shown in Figure 2. The preamble is common to both the sequential and parallel codes (Figure 2(a)). 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. An appropriate indexing function for two-dimensional arrays in C/C++ is assumed.

Conceptually, we can view an optimization as a change in the type of the shared object for the lifetime of the scope. The current set of available behaviours is summarized in Table 2. As an example of per-object flexibility, three different data sharing optimizations are applied to

Scoped Behaviour	Description
Owner-computes	Threads access only co-located data.
Caching for reads	Create local copy of data.
Release consistency	Buffer write accesses.
Special-purpose data movement	Used with owner-computes for specific applications (e.g., stencils in 2-D diffusion simulation).

Table 2. Some Scoped Behaviours

the sequential code in Figure 2(b) to create the parallel code in Figure 2(c). Specifically:

1. `NewBehaviour(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 co-located with a portion of `mA`'s distributed data actually enter the `while`-loop and iterate over their local data. Also, when `dotProd()` is called, a type constructor for `GVOwnerComputes` returns a C-style pointer to the local data so that the function executes with maximum performance.

   Although some changes to the source code are required to apply owner-computes, they are relatively straightforward. Other work partitioning strategies, that do not use the special methods provided by Aurora, are allowed, but owner-computes is both convenient and efficient.

2. `NewBehaviour(mB, GVReadCache, int)`: To automatically create a local copy of distributed vector `mB` at the start of the scope, since it is read-only and re-used many times, its type is changed to `GVReadCache`.

   The scoped behaviour of a read cache also includes a type constructor so that `dotProd()` can be called with C-style pointers that point to the cache. Note that no lexical changes to the loop's source code are required for this optimization.

3. `NewBehaviour(mC, GVReleaseC, int)`: To reduce the number of update messages to elements of distributed vector `mC` during the computation, its type is changed to `GVReleaseC`.

   Within the scope, the overloaded operators batch the updates into a per-target address space buffer and messages are only sent when the buffer is full or when the scope is exited. Also, multiple writers to the same distributed vector are allowed. No lexical changes to the source code are required.

The result of this heterogeneous set of optimizations is that the nested loops can execute without remote data accesses and the parallel program can use the same efficient `dotProd()` function as in the sequential program.

4.2 Discussion: Programming in Aurora

The typical methodology for developing Aurora applications consists of three main steps. First, the code is ported to Aurora. Shared arrays and shared scalars are converted to `GVector`s and `GScalar`s. Although the default immediate access policy can be slow, its performance can be optimized after the program has been fully debugged.

Second, the work is partitioned among the processors and threads. Owner-computes and SPMD-style parallelism are common and effective strategies for many applications. However, the application programmer may implement other work partitioning schemes.

Lastly, 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 `NewBehaviour` macro. Sometimes, straightforward changes to the looping parameters are needed for owners-computes. For example, in the matrix multiplication program, owner-computes can be applied 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. The new optimization strategy uses more resources for read caches than the original strategy, but, since `mC` is being updated, it is perhaps a more conventional application of owner-computes. Reverting back 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.

5 Scoped Behaviour

Scoped behaviour is a change in the implementation of selected methods for the lifetime of a language scope.

For the Aurora programmer, scoped behaviour is how an optimization is applied to a shared-data object. For the system and class designer, scoped behaviour is an interface between collaborating classes that changes the implementation of the selected methods. Some of the ideas behind scoped behaviour have been explored as part of the handle-body and envelope-letter idioms in object-oriented programming [Cop92] (to be discussed

(a) Scoped Behaviour Macro

```
#define NewBehaviour( XX, YY, ZZ ) \                  // Macro provided by aurora.H
  GPortal<GVector<ZZ> > AU_ ## XX( XX ); \
  YY<ZZ> XX( AU_ ## XX );

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 constructor
}; // GPortal
```

(b) Source Code

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


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

(c) After Standard Preprocessor Pass

```
{ // 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;                   // Immediate update (still)
```

Figure 3. Aurora's Scoped Behaviour Macro

further in Section 6.1). Scoped behaviour builds upon these ideas.

5.1 Language Scopes and Scoped Behaviour Objects

The main motivation for using language scopes to define the context of scoped behaviour is to exploit the property of *name hiding*. In block-structured languages, an identifier can be re-used within a nested language scope, thus hiding the identifier outside of the scope.

Instantiations of a class that are designed to be used within a language scope, and which hide objects outside the scope, are called *scoped behaviour objects*.

5.2 Implementing Scoped Behaviour

As shown in Figure 3(a), Aurora provides the scoped behaviour macro `NewBehaviour` and the class template `GPortal` via a header file. Figure 3(b) shows the original programmer's source code and Figure 3(c) shows the code after the standard preprocessor of the C++ compiler has expanded the macro. Again, the code is shown side-by-side for comparison.

The `NewBehaviour` macro is parameterized by the name of the original shared-data object, the type of the new scoped behaviour object, and the type of the vector elements.[1] The macro instantiates two objects. The first object, `AU_vector1`, is of type `GPortal`. Its sole function is to cache a pointer to the original object, which is passed as a constructor argument, and then pass it along to the scoped behaviour object's constructor. The second object, the scoped behaviour object `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 behaviour object can mimic or change the functionality of the original shared-data object.

We will discuss the implementation of these classes in more detail in Section 6, but we provide an overview of the basic ideas.

Since the scoped behaviour object has the same name as the original `vector1`, the compiler will generate

[1] Note that it is a multi-line 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 (i.e., the new `vector1` is passed a reference to itself, instead of to the original object), so an intermediary object is required. Fortunately, the macro hides the existence of the intermediary object.

the loop body code according to class `GVReleaseC` instead of the original object's class. However, the user's source code does not change. Even though the original and scoped behaviour objects collaborate to implement scoped behaviour, we can conceptualize it as temporarily changing the type of the original object. The `NewBehaviour` macro helps to hide this abstraction. Note that source code outside of the context of the optimization continues to refer to the original `GVector`. Therefore, immediate update remains the default behaviour outside of the scope, illustrating per-context flexibility.

The class template `GVReleaseC` is designed to behave exactly like `GVector`, except that the overloaded operators now buffer updates to the vector elements. Read accesses to the vector continue to be performed immediately, even if the data is remote. Thus, the class of a scoped behaviour object can selectively redefine behaviour on a method-by-method and operator-by-operator basis.

Also, since `vector1` is a new object within the scope, dynamic run-time actions can be associated with the various constructors and the destructor. In particular, the destructor flushes the update buffers to the vector so that all updates are guaranteed to be performed when the scope is exited.

Although this description has centered on a particular class, the basic scoped behaviour technique can be applied to a variety of classes and objects. The owner-computes, caching for reads, and other behaviours use the same `NewBehaviour` macro and are based on the same design principles.

Of course, the basic ideas behind the implementation of scoped behaviour are not new. The notion of nested scopes is fundamental to block-structured sequential languages. The association of data movement actions with C++ constructors and destructors is also not new (for example, in ABC++). However, scoped behaviour is unique in that it coordinates the interaction of different classes to create per-object and per-context behaviours.

5.3 Advantages and Disadvantages

The advantages of scoped behaviour include:

1. *Standards-based implementation.* Scoped behaviour can be implemented within standard C++ as a preprocessor macro. The class library, to be discussed in the next section, is also standard C++.

2. *Flexibility of experimentation.* Scoped behaviour makes it easy to add, modify, and remove behaviours with minimal or no lexical source code changes.

3. *Flexibility of implementation.* The compile-time aspect of scoped behaviour allows the compiler (and implementor) to generate behaviour-specific code based on different classes. The run-time aspect of scoped behaviour allows dynamic behaviour, such as data movement and interactions with the run-time system, to be associated with constructors and destructors.

A disadvantage of scoped behaviour is that, since it is a programming technique instead of a first-class compiler feature, it cannot access the compiler's symbol table for high-level analyses. A more general disadvantage is that, since the run-time behaviour depends on constructors and destructors with static invocation points, it cannot be directly ported to a language like Java [Sun96]. Java is a garbage-collected language and the current definition does not have destructors in the same sense as C++.

Compared to some other DSM and DSD systems, scoped behaviour has safety and performance benefits.

For example, `GVReleaseC` has been explicitly implemented with a constructor that takes a parameter of type `GVector&`. Therefore, programming errors involving incompatible objects, such as trying to use release consistency with normal C++ arrays, will result in compile-time errors. More generally, as with all object-oriented systems, methods are invoked on objects and thus it is impossible to pass the wrong shared-data object as a function call parameter. Also, the automatic construction and destruction of scoped behaviour objects make it impossible for the programmer to omit a required data movement action at the end of a context. Non-object-oriented function libraries may only be able to catch these forms of errors at run-time, if at all.

As with some other systems, performance benefits can arise from exploiting high-level data access semantics. For example, `GVReadCache` is intended for data that is read-only and where most of the elements will be accessed during the context. Therefore, Aurora can read the data in bulk rather than demanding-in each portion of the data with a separate data movement action. Also, `GVReleaseC` is intended for data that is updated but not read. Therefore, unlike some other systems, Aurora can avoid the overhead of demanding-in the remote data before overwriting it.

6 Shared-Data Class Library

In this section, we 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 behaviour.

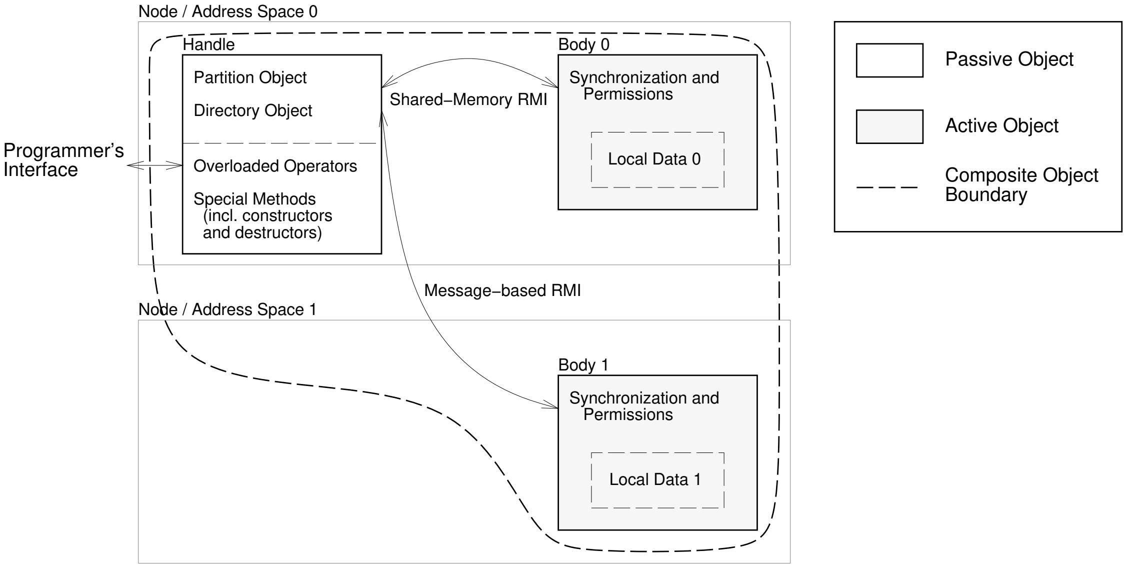

Figure 4. Handle-Body Composite Objects

6.1 Handle-Body Composite Objects

The main architectural feature of the shared-data class library is the use of the handle-body idiom to create *composite objects* [Cop92, OEPW96] for shared data (Figure 4). The *handle object* defines the programmer's interface to the shared data. The *body object* (or objects) contain the actual data.

The extra level of indirection afforded by a composite handle-body approach allows for:

1. *Data distribution.* A distributed vector is a set of body objects and each body object can be located in a different address space or on a different physical 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 4 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, the distributed data can be accessed 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. *Cheap parameter passing of shared data.* Only handles are passed across function calls; the data in the bodies are not copied. Handles can also be passed between address spaces, if desired, since 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 overheads of indirection can be avoided in controlled ways through type constructors that return C-style pointers.

The current implementation of Aurora creates handles as passive (i.e., regular) C++ objects. However, each individual body is implemented as an active object, which is useful for implementing any necessary synchronization behaviour. Handle and body interact using remote method invocations. The run-time system automatically selects between shared-memory and message-based communication mechanisms for transmitting RMIs.

6.2 Class Hierarchy for Handles

Since most of the data sharing functionality is implemented in the handles, this discussion will focus on the handle classes. Briefly, however, the body classes support `get()` and `put()` data access methods, including batch update and block-read variations. For the current data sharing optimizations in Aurora, this simple functionality is all that is required.

Figure 5 is a diagram of the main classes in the class hi-

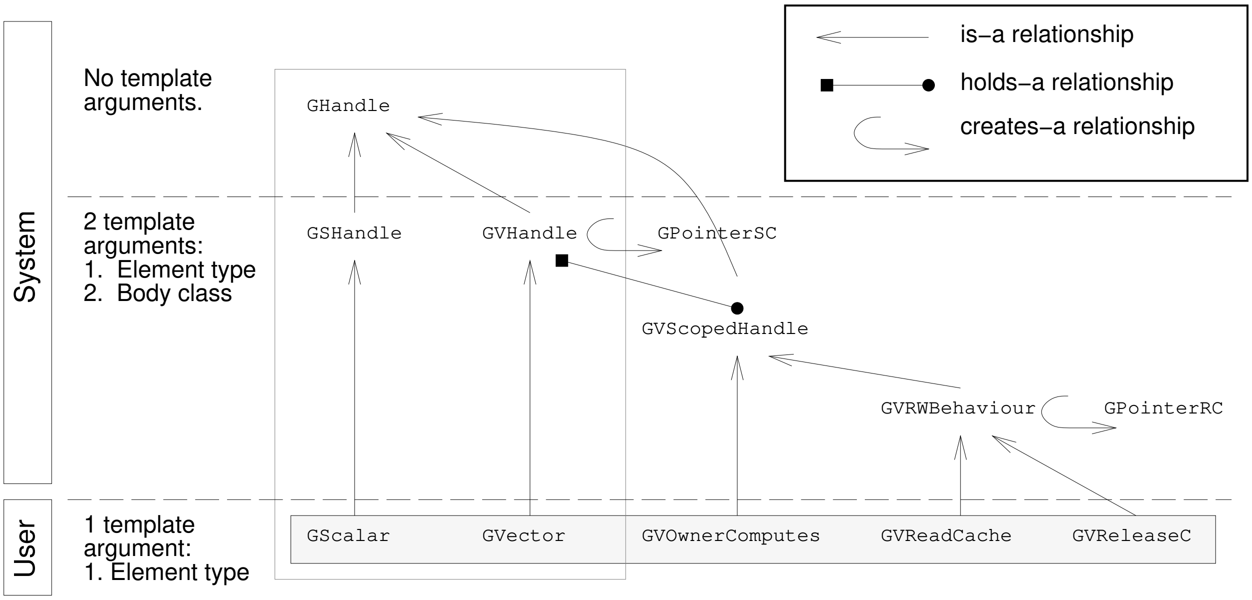

Figure 5. Class Hierarchy for Handles

erarchy for shared-data handles.[2] Aside from the names of the classes, the diagram shows the relationship between classes. The *is-a* relationship is the usual notion of inheritance. For example, class `GHandle` is the base class for all handles. Common access methods are factored into the base class. The *holds-a* relationship exists when a class contains a pointer (or pointers) to an instance of another class. This is used, for example, to allow one object to access the internal state of another object. The *creates-a* relationship exists when at least one of the methods of a class returns an object of another class. For example, an overloaded subscript operator (i.e., `operator[]`) can return an object which encodes information about a specific vector element [Cop92].

We can also distinguish the classes by the way they are, or are not, templated. Class `GHandle` is not templated in order to simplify the implementation of mechanisms that only require limited functionality from a handle. For example, querying about the number of vector elements does not require knowledge about template arguments. However, the most important class templates for the system implementor are parameterized by both the data element type and the class of the body object.

In general, the application programmer is only expected to use the classes with a single template argument for the data element type (labelled "User" in Figure 5 and highlighted in gray). These classes hide the more complex templating and class hierarchy considerations that the "System" must deal with.

For data sharing using immediate access, the important classes are `GSHandle` and `GVHandle` (shown inside the box in Figure 5). These classes encapsulate member data to keep track of the body or bodies.

Figure 6 provides a more detailed look at the interfaces for the classes that implement the shared vector. Class `GHandle`, which is not templated, is a convenient base class within which to implement methods common to all handles. Class `GVector` does little more than specify the specific body class (i.e., `LVector`) for the second template argument to `GVHandle` and call the appropriate constructors.

Most of the functionality for the shared vector is implemented by class `GVHandle`. In particular, the overloaded subscript operator 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 sub-expressions. Since the actual data for a term may be a remote shared data element, the temporary object points to the body object with the data. Class `GPointerSC` has data members to store the vector index and a pointer to the specific body object with that element. Reading from or writing to the vector element invokes the appropriate type constructors and the overloaded assignment operator of `GPointerSC`, resulting in an immediate remote memory access.

[2] The notation is based on Booch [Boo91], but with some simplifications and changes to better suit this presentation.

```
// Base class. Not templated.
class GHandle
{
  private:
    int numElements;                                        // Number of vector elements
    // ...other data members...
  public:
    // ...various constructors and destructor...
    int size() { return numElements; }                      // Common access method
    // ...other methods...
}; // GHandle (System)


// Template argument C_Data is the element type; C_LV is the body class.
//   Classes GVScopedHandle, Partition, Directory, GPointerSC are provided by Aurora.
template <class C_Data, class C_LV>
class GVHandle : public GHandle                             // is-a GHandle
{
    // GVScopedHandle needs access to internal state (for holds-a)
    friend GVScopedHandle<C_Data, C_LV>;
  protected:
    Partition<MAX_LOCALS> partition;                        // Distribution strategy
    Directory<C_LV> directory;                              // Location of body object(s)
    // ...other data members...
  public:
    GVHandle( int numElements );                            // Construct with size of vector
    ~GVHandle();
    GPointerSC<C_LV, C_Data> operator[] ( int index );      // Immediate data access (creates-a)
    // ...other methods...
}; // GVHandle (System)


// Template argument C_Data is the element type; LVector (provided by Aurora) is the body class.
template <class C_Data>
class GVector : public GVHandle<C_Data, LVector<C_Data> >   // is-a GVHandle
{
  public:
    GVector( int numElements ) :                            // Construct with size of vector
          GVHandle<C_Data, LVector<C_Data> >( numElements ) {}
    ~GVector();
    // ...inherits operator[] and other methods...
}; // GVector (User)
```

Figure 6. Interface for Shared Vector: `GVector`

```
// Template argument C_Data is the element type; C_LV is the body class.
//   Remember that I am a friend of GVHandle.
template <class C_Data, class C_LV>
class GVScopedHandle : public GHandle                                    // is-a GHandle
{
  protected:
    GVHandle<C_Data, C_LV> * origHandle;        // To access internal state of original object (holds-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)


// 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 GVRWBehaviour : public GVScopedHandle                      // is-a GVScopedHandle
{
  protected:
    Cache<C_Data, C_LV> * readCache;                                // Configurable read cache
    BatchWrite<C_Data, C_LV> * updateBuf[MAX_LOCALS];     // Configurable buffer for release consistency
    // ...other data members...
  public:
    GVRWBehaviour( GVHandle<C_Data, C_LV> & gv ) :                 // Construct with original handle
          GVScopedHandle<C_Data, C_LV> >( gv ) {}
    ~GVRWBehaviour();                                   // 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 );    // Data access via cache/buffer (creates-a)
    // ...other methods...
}; // GVRWBehaviour (System)


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

Figure 7. Interface for Release Consistency Scoped Behaviour: `GVReleaseC`

6.3 Data Sharing Optimizations: Scoped Behaviour Objects

For the data sharing optimizations, the parent class `GVScopedHandle` extracts and maintains information about the internal state of a given `GVHandle`, as per the holds-a relationship (Figure 7). This functionality is an important part of implementing scoped behaviour. The partition and directory objects of the `GVHandle` are not copied, thus reducing the construction costs of a scoped behaviour object.

Class `GVOwnerComputes`, in its constructor, uses the extracted internal state to determine the address of the body object's data. Therefore, `GVOwnerComputes` can return a C-style pointer from the appropriate type constructor and from the overloaded subscript operator. As previously discussed, `GVOwnerComputes` also defines special functions to support easy iterating over the local data.

Class `GVRWBehaviour` can, optionally, create a read cache for shared data and create update buffers to shared data (Figure 7). Classes that derive from `GVRWBehaviour` explicitly configure the caching and buffering options. The overloaded subscript operator in `GVRWBehaviour` 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, then `GPointerRC` is configured to access data from the cache instead of from the remote body. Second, if the update buffers are enabled in `GVRWBehaviour`, then `GPointerRC` is configured to store updates in the buffer rather than initiate a remote memory access. `GVRWBehaviour` creates the buffers on demand. Depending on the configuration of the cache and buffers, `GPointerRC` will access shared data appropriately.

Therefore, the constructor of class `GVReadCache` calls the appropriate `GVRWBehaviour` methods to create and load the read cache. Thus, when the subscript operator for `GVReadCache`, which is inherited from the parent class, creates a `GPointerRC` object, it will always access the cache. `GVReadCache` also defines a type constructor to return a C-style pointer to the cache.

Similarly, class `GVReleaseC` calls the appropriate `GVRWBehaviour` constructor and enables the use of update buffers (Figure 7). Thus, when the subscript operator for `GVReleaseC`, which is inherited from the parent class, creates a `GPointerRC` object, it will always use the buffers. The destructor for class `GVRWBehaviour` makes sure all buffers are flushed.

7 Extending the Library

Within the class hierarchy, new data sharing optimizations can be implemented. We consider a trivial but illustrative example. For example, a new class could both cache data for reading *and* buffer updates. The new class would derive from `GVRWBehaviour`. The new class's constructor creates the read cache and also enables the update buffers. The `GPointerRC` objects created by the new class would always read from the cache and always buffer updates. By default, updates are also mirrored in the cache. Admittedly, this "new" data sharing optimization is easy to add because of the design and existing functionality of `GVRWBehaviour` and `GPointerRC`, but the basic techniques can be used for more complex additions to the library.

There are three main techniques for extending the library of data sharing optimizations. The techniques can also be combined.

1. *New classes.* Define new classes for partition, directory, body, and pointer objects.

   Currently, only a block-distributed partition object is implemented. If a cycle-distributed object is required in the future, a new partition class could abstract the distribution details. Finally, as we have seen, classes like `GPointerSC` and `GPointerRC` are useful for defining new memory access behaviours.

2. *New methods.* Inherit from a parent class, then add new scoped behaviour with new methods.

   For example, `GVOwnerComputes` adds new methods for iterating over local data.

3. *Re-define methods.* Inherit from a parent class, then re-define behaviour through constructors, the destructor, methods, operators, and type constructors.

   For example, `GVReleaseC` relies on its parent class for most of its functionality. `GVReleaseC` merely configures the update buffers appropriately in its constructor.

8 Performance

To date, we have experimented with three Aurora programs [Lu97]. The programs are matrix multiplication (Figure 2), a 2-D diffusion simulation, and Parallel Sorting by Regular Sampling (PSRS) [SS92, LLS+93]. Recent performance results are shown in Table 3. Speedups are computed against C implementations of the same algorithm (or against quicksort in the case of the parallel

Program	Data Set	Network	Speedup		
			2 PEs	4 PEs	8 PEs
Matrix Multiply	704×704 (175 sec. seq.)	Fast Ethernet	1.85	3.51	6.40
	512×512 (65.8 sec. seq.)	Fast Ethernet	1.79	3.37	5.89
2-D Diffusion	1526×1526, 32 time-steps (47.8 sec. seq.)	Fast Ethernet	1.27	2.13	3.86
	1024×1024, 32 time-steps (20.3 sec. seq.)	Fast Ethernet	1.07	1.91	3.45
PSRS	10 million keys (60.4 sec. seq.)	Fast Ethernet	n/a	2.24	3.72
	6 million keys (33.9 sec. seq.)	Fast Ethernet	1.21	2.05	3.22

Table 3. Aurora Programs on a Network of Workstations

sort). In particular, the sequential implementations do not suffer from the overheads of either operator overloading or scoped behaviour.

The distributed-memory platform used for these experiments is a cluster of PowerPC 604 workstations with 133 MHz CPUs, 96 MB of main memory, and a single, non-switched 100 Mbit/s Fast Ethernet network. The software includes IBM's AIX 4.1 operating system, AIX's pthreads, and the MPICH (version 1.0.13) [DGLS93] implementation of MPI.

Two trends can be noted in the performance results. First, for these three programs, additional processors improves speedup, albeit with diminishing returns. Second, as the size of the data set increases, the overall granularity of work, and thus speedup, also increases.

Contention for the single network and a reduced granularity of work can account for the diminishing returns for more processors with a fixed problem size. For example, since the read cache's data requirements are constant per-processor, communication costs and network contention grows when replicating vector `mB` in matrix multiplication. Communications costs under contention also account for the overheads in the parallel sort program, since the algorithm includes a key exchange. For the 2-D diffusion simulation, the granularity of a time-step before a barrier quickly falls to below one second as processors are added. Fortunately, if the problem size increases, the computation's overall granularity also increases resulting in better absolute speedups.

The performance of Aurora programs on this particular hardware platform is encouraging, but there remains two important avenues for future work: different network technology and new scoped behaviours. An 155 Mbit/s ATM network has been installed on the platform, but it is not yet fully exploited by the run-time system. However, early experience indicates that the additional bandwidth and improved contention characteristics of ATM will benefit Aurora programs. Also, there is currently no overlap between communication (for reads) and computation in the existing scoped behaviours. For simplicity, `GVReadCache` loads all of the data before allowing computation to continue. Using the techniques described in this paper, the library of scoped behaviours will be extended to better hide the read latency of the distributed-memory hardware.

9 Discussion and Related Work

Distributed data sharing is an example of a problem domain where per-object and per-context optimization flexibility is desirable. The data access behaviour of a shared-data object can change depending on the loop or program phase, so a single data sharing policy is often insufficient for all contexts. In general, optimization flexibility can be supported through compiler annotations or a run-time system interface, but scoped behaviour offers advantages in terms of engineering effort, safety, and implementation flexibility.

Since Ivy [Li88], the first DSM system, a large body of work has emerged in the area of DSM and DSD systems (for example, [BCZ90, BKT92, BZS93, SGZ93, JKW95, ACD+96]). Related work in parallel array classes (for example, [LQ92]) has also addressed the basic problem of transparently sharing data.

Different access patterns on shared data can be optimized through type-specific protocols and run-time annotations. Both Munin [BCZ90] and Blizzard [FLR+94] provide protocols customized to specific data sharing behaviours. Run-time libraries, such as shared regions [SGZ93], SAM [SL94], and CRL [JKW95], associate coherence actions with access annotations (i.e., 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 behaviour objects 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 the parametric shared region (PSR) mechanism of ABC++. However, there are some significant differences between Aurora's shared-data objects and PSRs. First, Aurora allows distributed vectors to be partitioned between different address spaces to improve scalability and to support owner-computes using multiple nodes. A PSR has single home node, therefore shared data cannot be partitioned and owner-computes cannot be used within a PSR. Second, Aurora uses operator overloading and pointer objects, which gives the system more flexibility to generate behaviour-specific code, and to optimize the read and write behaviour of shared data separately. Aurora can also return C-style pointers to shared data under controlled circumstances. The data in a PSR is always accessed using C-style pointers, which is efficient, but it does not allow the system to selectively intervene in data accesses. Lastly, Aurora supports multiple writers to the same distributed vector object, which can be important for performance [ACD+96], while PSRs only allow a single writer.

10 Concluding Remarks

Researchers have explored a variety of different implementation techniques for DSM and DSD systems. The Aurora DSD programming system is an example of a software-only implementation that uses data sharing optimizations to achieve good performance on a set of parallel programs.

What distinguishes Aurora from other DSM and DSD systems is its use of scoped behaviour as an interface to a set of data sharing optimizations. Scoped behaviour supports per-context and per-object flexibility in applying the optimizations. This novel level of flexibility is particularly useful for incrementally tuning multi-phase parallel programs and programs in which different shared objects are accessed in different ways. The performance of Aurora is encouraging and future work will explore new data sharing optimizations and how they can exploit different network performance characteristics.

Scoped behaviour can be implemented in standard C++ without special compiler support and it offers important safety benefits over typical run-time libraries. The technique appears to be a viable approach for supporting this form of optimization flexibility.

11 Acknowledgments

Thank you to Ben Gamsa, Eric Parsons, Karen Reid, Jonathan Schaeffer, Ken Sevcik, Michael Stumm, Greg Wilson, Songnian Zhou, and the anonymous referees for their comments and support during this work. Thank you to the Department of Computer Science and NSERC for financial support. Thank you to ITRC and IBM for their support of the POW Project.

References

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AG96] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[BCZ90] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 1990 Conference on Principles and Practice of Parallel Programming*. ACM Press, 1990.

[BKT92] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[Boo91] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[BZS93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. 38th IEEE International Computer Conference (COMPCON Spring'93)*, pages 528–537, February 1993.

[Cop92] J.O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison–Wesley, 1992.

[DGLS93] N.E. Doss, W.D. Gropp, E. Lusk, and A. Skjellum. A Model Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1993.

[FLR+94] B. Falsafi, A.R. Lebeck, S.K. Reinhardt, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proc. Supercomputing '94*, pages 380–389, November 1994.

[GLL+90] K. Gharachorloo, D.E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GLS94] W.D. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[JKW95] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. 1988 International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[LLS+93] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19:1079–1103, 1993.

[LQ92] M. Lemke and D. Quinlan. P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications. In *Proc. CONPAR 92–VAPP V*. Springer-Verlag, September 1992.

[Lu97] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997. Available at http://www.cs.utoronto.ca/~paullu/.

[OEPW96] W.G. O'Farrell, F.Ch. Eigler, S.D. Pullara, and G.V. Wilson. ABC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[Pth94] Draft Standard for Information Technology—Portable Operating Systems Interface (Posix), September 1994.

[SGZ93] H.S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence. In *Proc. Symposium on Principles and Practices of Parallel Programming*, pages 229–238, May 1993.

[SL94] D.J. Scales and M.S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. 1st Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[SS92] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[Sun96] Sun Microsystems. *The Java Language Specification, Version 1.0*, August 1996. http://www.javasoft.com/doc/language_specification/.