See IEEE Concurrency, 8(3):63–73, July–September 2000
for the published version. This was the accepted version.

Implementing Scoped Behaviour for Flexible Distributed Data Sharing

Paul Lu
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2H1
Canada
paullu@cs.ualberta.ca

Abstract

Distributed-memory hardware platforms, such as a network of workstations, are attractive because of their ubiquitousness and good price-performance. However, there are high communications overheads associated with sharing data between distributed memories. While message-passing programming systems provide the greatest low-level flexibility to optimize the overheads, shared-data systems provide a higher level of abstraction. Ideally, one would like to have both a high level of abstraction *and* the flexibility to optimize a program for each data-sharing pattern and for each portion of the source code (i.e., context), such as a particular loop or phase.

A novel technique to support this form of optimization flexibility is *scoped behaviour*. In the Aurora distributed shared data system, the programmer instantiates shared-data objects and uses scoped behaviour to incrementally tune applications on a per-object and per-context basis. We detail how a class library implements shared-data objects as abstract data types and how scoped behaviour implements the optimizations within standard C++. Using a network of workstations connected by an ATM switch, we show that the performance of Aurora is comparable to message passing (using MPI) for three applications.

Keywords: distributed-memory computing, shared data, data-sharing patterns, optimizations, scoped behaviour, network of workstations

1 Introduction

Parallel and distributed computing is a diverse area of research and practice [1]. Many different hardware architectures have been proposed, but networks of workstations and clusters have recently garnered a lot of 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. Since the data-sharing policies of an application determine how often, when, and what mechanisms are used for communications, they have a large impact on performance and must be optimized.

The flexibility to tune a distributed-memory application varies depending on what kind of parallel programming system is used. Parallel applications can be developed using parallel languages (e.g., High Performance Fortran), libraries (e.g., LAPACK), run-time systems (e.g., 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; 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 makes explicit when data is sent and received. With sufficient (and often substantial) programming effort, a message-passing program can be highly tuned. Ideally, one 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 behaviour*, that provides a high degree of flexibility in applying an optimization within a high-level parallel programming system.[1] We begin with a discussion of some of the existing approaches to distributed data sharing. Then, we describe how scoped behaviour is used by application programmers, how the technique is implemented, and how three optimized applications perform on a network of workstations.

The implementation of scoped behaviour is noteworthy because it does not require any language extensions to C++, nor does it require special compiler support. However, the implementation details in Section 4 are heavy on object-oriented design, thus it may not interest all readers at first. Section 5 continues the more general-interest discussion of how the three applications achieve comparable performance to message passing.

1.1 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 is determined by 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 and widespread 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 it more convenient and less error-prone to use. Consequently, a variety of software-based logically-shared systems for distributed-memory platforms have been developed. The systems emphasize usability, but they improve performance through a variety of optimizations. Broadly speaking, there are *distributed shared memory* (DSM) [3, 4] and *distributed shared data* (DSD) [5, 6] systems. Table 1 compares the characteristics of "typical" DSM and DSD systems.

At one end of the shared-data spectrum, DSM systems use software to emulate hardware-based shared memory. Typically, DSM systems are based 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 (e.g., `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

[1] This article is an expanded version of an earlier paper [2],

	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

Table 1: Distributed Shared Memory (DSM) versus Distributed Shared Data (DSD)

is transparent to the programmer in the same way that virtual memory faults into the operating system are also transparent. But, instead of using a traditional backing store, the DSM run-time system communicates with other processor nodes in order to update, invalidate, and otherwise manage the data in a shared page. Often, a single data-sharing policy is used for all shared pages, although per-page policies are supported by some DSM systems.

At the other end of the spectrum, DSD systems treat shared data as an *abstract data type* (ADT). 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 may 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 since ADTs are not dependent 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 read and writes, so the programmer can select the ADT with the most optimized policy for a given pattern. Consequently, the unit of sharing and the sharing policy can be selected to match the form and function of the particular data.

1.2 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, since 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.

In DSD systems, false sharing is avoided by managing independent data as separate objects. And, as

(a) Original Loop	(b) Optimized Loop Using Scoped Behaviour

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

previously discussed, 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 is used as the input of another function, the same shared data may be 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 behaviour. As the programmer's interface for specifying a data-sharing optimization, scoped behaviour allows each shared-data object and each portion of the source code (i.e., context), to be optimized independently of other objects and contexts.

2 Aurora: Flexibility Using Scoped Behaviour

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 from any processor node. In keeping with an ADT 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 the details of resource allocation and deallocation, 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 allow the shared data to 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 behaviour to provide:

- *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 implementation details are discussed later on, but we first cover the higher-level programming abstractions of Aurora.

Scoped Behaviour	Description
Owner-computes	Threads access only co-located data.
Caching for reads	Create local copy of data.
Release consistency	Buffer write accesses.

Table 2: Some Scoped Behaviours

2.1 Example: A Simple Loop

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 concrete type [7] 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 processor nodes. Therefore, `vector1` is a vector object with 1024 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 (i.e., `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 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, shared data is read from and written to synchronously, even if the data is on a remote node, since that data access behaviour 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, then the loop can be optimized by using release consistency [8]. Basically, all writes are buffered and updates to the vector will be done at a later time, instead of synchronously. Three new elements are required to use scoped behaviour to specify the optimization (see Figure 1(b), shown side-by-side with Figure 1(a) for easy comparison): 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 `NewBehaviour` macro specifies that the release consistency optimization should be *applied* to `vector1`. Upon re-compilation, and without any changes to the loop code itself, the *behaviour* of the updates to `vector1` is changed within the language scope. The new behaviour uses buffers to batch the writes and automatically flushes the buffers when the scope is exited.

2.2 Example: Matrix Multiplication

We now examine a more complex example of using scoped behaviour. Consider the problem of non-blocked, 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 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.

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

(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

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 block-distributed data actually enter the `while`-loop and iterate over their local data. Note that `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. `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 allows `dotProd()` to 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`.

   As with the simple loop example, the overloaded operators batch the updates into buffers and messages are only sent when a 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 with far fewer remote data accesses than before. All read accesses are from a cache or local memory; all write accesses are buffered.

The high-level semantics of scoped behaviours can also be exploited 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 behaviours do contain extra semantic information. In particular, the read cache scoped behaviour specifies that *all* of vector `mB` in matrix multiplication is cached, therefore there is no need to transfer each unit of data separately. The multiple request messages can be eliminated if the data is streamed into each read cache via bulk data transfer. Of course, bulk data transfer is not unique to Aurora, but the high-level semantics and the flexibility of scoped behaviour provide a natural conceptual and implementation framework.

2.3 Discussion: 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. First, shared arrays and shared scalars are converted to `GVectors` and `GScalars`. Although the default synchronous access policy can be slow, its performance can be optimized after the program has been fully debugged. Second, 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. 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 `New-Behaviour` 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 behaviour makes it easier to experiment with different optimization strategies. 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. 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.

3 Scoped Behaviour

Scoped behaviour is a change in the interface of an ADT for the lifetime of a language scope. For the application programmer, scoped behaviour is how an optimization is applied to a shared-data object. For the system and class designer, scoped behaviour is a collaboration between classes that changes the implementation of selected methods.

3.1 Handle-Body Composite Objects

Some of the ideas behind scoped behaviour have been explored as part of the handle-body and envelope-letter idioms in object-oriented programming [7], the strategy design pattern [9], and parametric shared regions in ABC++ [10]. Scoped behaviour builds upon these ideas.

Composite objects, such as handle-body objects, are multiple objects that behave as if they were a single entity [7]. 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

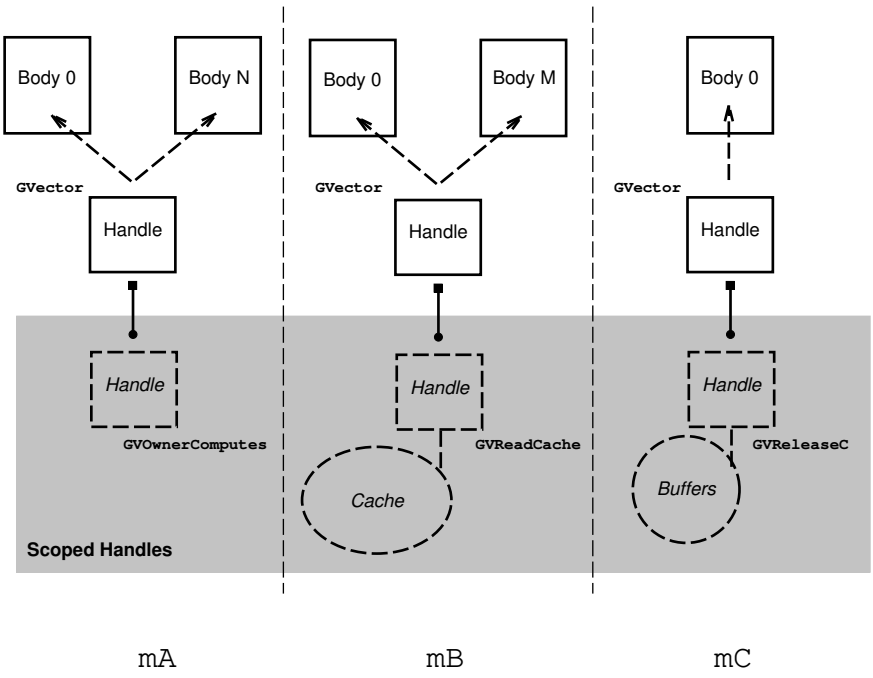

Figure 3: Composite Objects in Matrix Multiplication

---

way to support different ADT interfaces to the same encapsulated data. Depending on which handle is in use (i.e., in scope), the methods and behaviours will be different. The interaction between handle and body is discussed further in Section 4, but we first focus on how different handles interact.

3.2 Language Scopes and Scoped Handles

Language scopes are used to define the context of scoped behaviour in order to exploit the compile-time property of *name hiding* and the run-time properties of object creation. In many block-structured languages, an identifier can be re-used 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 called 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` have 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 behaviours. 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 (e.g., 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.

3.3 Handle References and Alternate Interfaces

As shown in Figure 4(a), Aurora provides the scoped behaviour macro `NewBehaviour` to help establish the reference from one handle to another. Figure 4(b) shows the original programmer's source code and Figure 4(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.

(a) Scoped Behaviour Macro

```
#define NewBehaviour( 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
```

(b) Source Code

```
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
vector1[ 0 ] = 1;              // Synchronous update
```

(c) After Standard Preprocessor Pass

```
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)
```

Figure 4: Aurora's Scoped Behaviour Macro

The `NewBehaviour` macro is parameterized by the name of the original shared-data object (`ORIG`), the scoped behaviour or type of the new scoped handle (`SB`), and the type of the vector elements (`TYPE`).[2] 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 functionality of the original shared-data object and the user's source code does not change. In other words, the programmer's interface of the ADT is changed without changing the encapsulated data or the user's source code.

Since 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` is designed to behave 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 behaviour as using the `NewBehaviour` macro to temporarily change the type of the original object.

Note that the source code outside of the context of the optimization continues to refer to the original

[2]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 semantics of the C++ standard. According to the standard, the new `vector1` is passed a reference to itself, instead of to the original object, so an intermediary `GPortal` object is required. Fortunately, the macro hides the existence of the intermediary object.

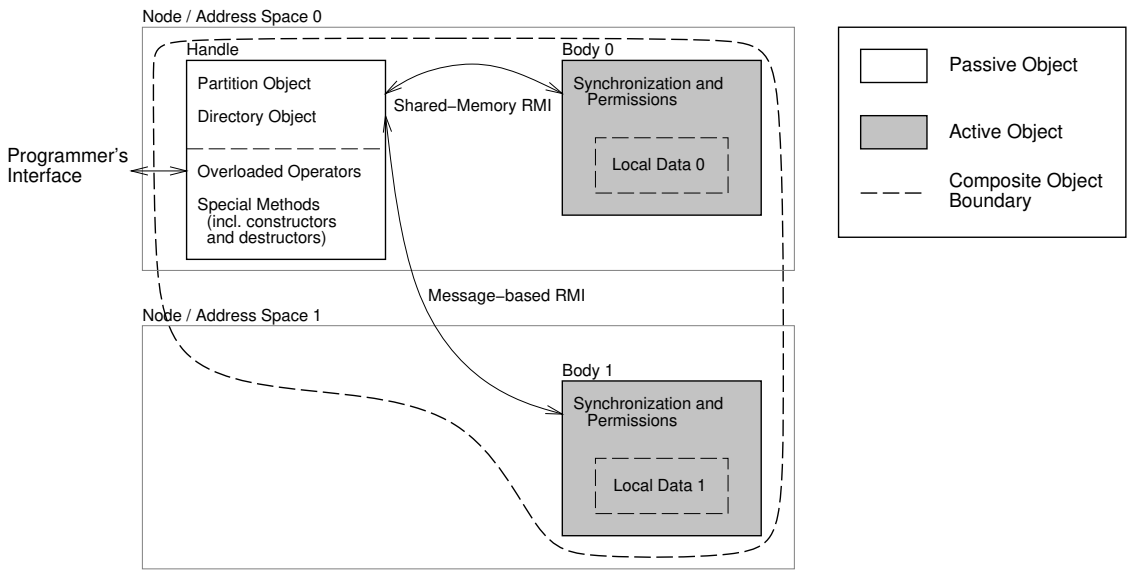

Figure 5: Shared-Data Composite Objects

`GVector`. Therefore, synchronous updates remain the default behaviour outside of the scope, illustrating per-context flexibility.

4 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. Again, the details presented in this section may be skipped upon first reading.

4.1 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 behaviour, the extra level of indirection between handle and body 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 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, 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.

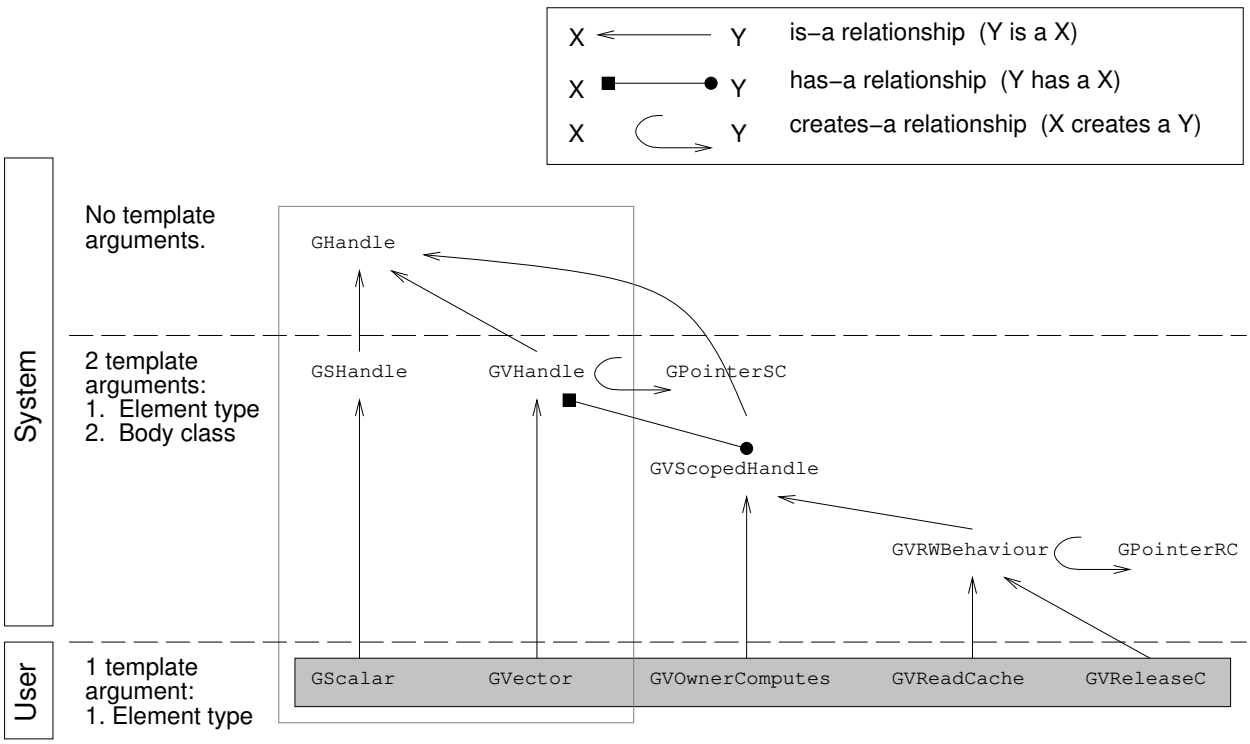

Figure 6: Class Hierarchy for Handles

3. *Cheaper 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 handle-body indirection can be avoided in controlled ways through *type conversion operators* that return C-style pointers.[3] 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, since `dotProd()` is expecting pointers as parameters, converts between a handle object and a pointer to the same data.

The current implementation of Aurora creates handles as regular C++ objects. However, each individual body is implemented as an active object, an object with its own thread of control, which is useful for implementing any necessary synchronization behaviour. 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++ [10]. The run-time system automatically selects between shared-memory and message-based communication (i.e., MPI) mechanisms for transmitting RMIs.

[3] Actually, the proper name is simply "conversion operator." We are more verbose in order to be more descriptive.

	GVector	Data-Sharing Optimizations		
		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

Table 3: Object Model for Handles

4.2 Class Hierarchy for Handles

Since most of the data-sharing functionality is implemented in the handles, this discussion will focus on the handle classes. Figure 6 is a diagram of the main classes in the hierarchy of shared-data handles.[4] In general, the application programmer is only expected to use the classes at the leaves of the hierarchy (labelled "User" and highlighted in gray). These classes hide the more complex templating and class hierarchy considerations that the "System" must deal with.

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," then an object of class Y contains a reference or pointer to an object of class X. With the right access control permissions, Y can access the internal state of X. Class Y can also call any method in class X in order to delegate functionality and behaviour. Note that, 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 has-a relationship is discussed further in Section 4.3.

The *creates-a* relationship exists when at least one of the methods of a class 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 (i.e., operator[]) can return a temporary object which 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 3). 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 lifetime of the object (i.e., 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 sub-expressions [7]. Reading from or writing to the vector element invokes the appropriate method (i.e., type conversion operator) and the overloaded assignment operator of GPointerSC,

[4]The notation is based on that of Grady Booch but with some simplifications and changes to better suit this presentation.

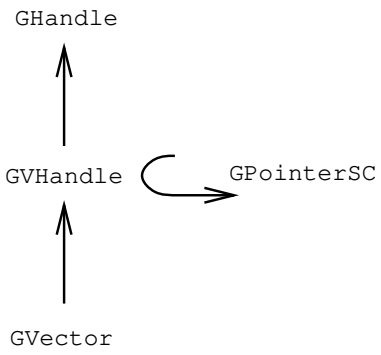

Figure 7: Class Hierarchy for `GVector`

resulting in a synchronous remote memory access. Further details and sample C++ code can be found elsewhere [2].

4.3 Data-Sharing Optimizations: Scoped Handles

The create-use-destroy object model is combined with handle references to implement the scoped handles (Table 3). 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.

Class `GVOwnerComputes`, in its constructor, uses the handle reference to determine the address of the local (i.e., 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 previously discussed in Section 2.2, `GVOwnerComputes` also defines special functions to support easy iterating over the local data.

Most of the read-write functionality of the caching for reads and release consistency behaviours are implemented within a cache and buffer manager class. Class `GVRWBehaviour` can, optionally, create a read cache for shared data and create update buffers to shared data (Figure 9). 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, then `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 `GVRWBehaviour` methods to create and load the read cache. Similarly, the constructor of class `GVReleaseC` calls the appropriate `GVRWBehaviour` method to enable the use of update buffers. The destructor for class `GVRWBehaviour` makes sure all buffers are flushed.

```
// 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`

```
// 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<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:
    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 );       // Pointer object to 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> & gv ) :             // Original handle via GPortal/NewBehaviour macro
          GVRWBehaviour<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 Behaviour: `GVReleaseC`

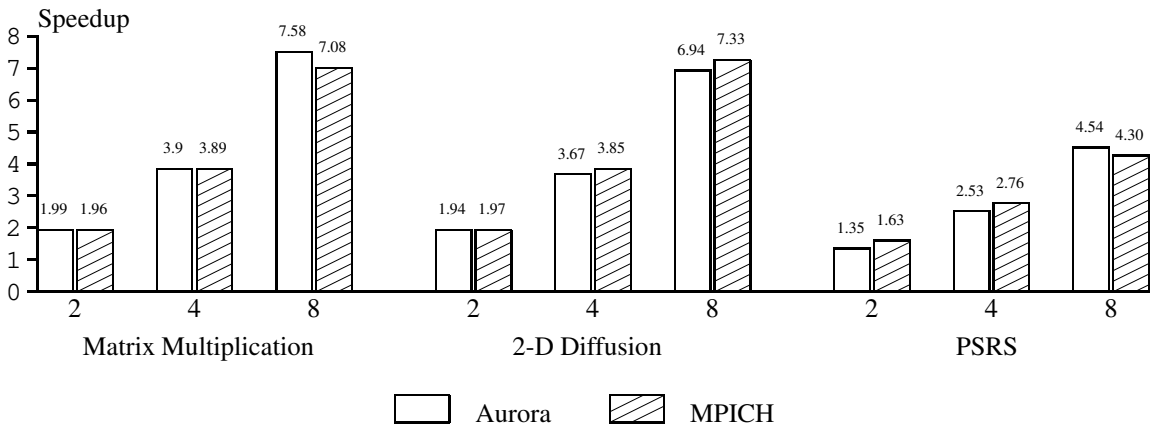

Figure 10: Speedups for Aurora and MPICH Applications: 2, 4, and 8 processors, 155 Mbit/s ATM

5 Performance Evaluation

We now examine the performance and data-sharing overheads of applications implemented using both Aurora and a MPI-based message-passing system. Aurora's performance is shown to be comparable to message passing for the three applications considered. We also detail how a particular scoped behaviour optimization allows Aurora to exceed the performance of basic message passing for one application.

The hardware platform used for these experiments is a cluster of PowerPC 604 workstations with 133 MHz CPUs, 96 MB of main memory, and a 155 Mbit/s 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. Note that MPICH is used as part of the run-time 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, we 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 2-D 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 [11]. The Aurora and MPICH implementations share much of the sequential portions of the source code. The respective speedups for up to 8 processors are shown in Figure 10. 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 behaviours, is used for both phases, but the function is called with different shared-data objects as parameters. In Phase 1, $P \leftarrow Q \times R$ is computed. In Phase 2, $R \leftarrow Q \times P$ is computed. In contrast to Figure 2, the owner-computes scoped behaviour is applied to both `mA` and `mC` and the read cache scoped behaviour is applied to `mB`. This makes it easier to evaluate the overheads of the read cache since the owner-computes optimization does not involve any communication overhead. Although the specific matrix computation is synthetic, it is designed to demonstrate how different scoped behaviours can be applied to a shared-data object in different computational phases.

For matrix multiplication, Aurora achieves higher speedups than MPICH, especially for 8 processors,

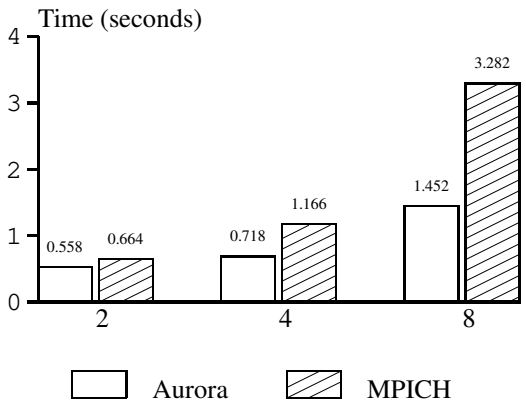

Figure 11: Data-Sharing Overheads in Matrix Multiplication: 2, 4, and 8 processors

due to the optimizations provided by the read cache scoped behaviour. We isolated and measured the data-sharing overheads (Figure 11). As previously discussed, the high-level semantics of using read caches can be exploited by using bulk data transfer protocols to fill the caches. The real-time overheads of Aurora 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. Note that the performance advantage of Aurora versus MPICH increases with the number of processors. As the number of processors increases, the opportunities for congestion (e.g., n processors sending to one processor) 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 WANs, but it is not optimal for dedicated LANs 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 behaviour would simply use the new functionality. Scoped behaviour 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 2-D diffusion application simulates the diffusion of matter over time by computing a 9-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 2-D 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 timestep because the exchange of data can be an implicit synchronization. With a shared-data approach, an explicit and separate barrier, with associated overheads, is typically required to prevent the premature transfer of data.

PSRS is a multi-phase parallel sorting algorithm that includes a communication-intensive data exchange phase for keys [12], which limits the speedups. The MPICH version of PSRS achieves higher speedups than the Aurora version for the 2 and 4 processor cases. In the data exchange phase of the PSRS algorithm, the amount of data to be communicated can vary depending on the specific input data [11]. This is in contrast to the "always exchange all the data" semantics of a read cache in matrix multiplication. So, the previous

scoped behaviour cannot be used and a new data-sharing optimization is not yet implemented.

Interestingly, in the 8 processor case, Aurora outperforms MPICH. Although there is no specific scoped behaviour 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 may 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. A more detailed performance evaluation is in progress.

6 Discussion and Related Work

One disadvantage of the scoped behaviour approach is that each different behaviour requires additional implementation effort. Of course, it is the system designer who must implement the new scoped behaviours, and not the application programmer. Fortunately, data-sharing patterns do reappear in different contexts [3, 12] and our own experience is that scoped behaviours are highly re-usable. 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 behaviours is small, the behaviours can be combined on a per-context and per-object basis to support a variety of optimization strategies. And, it is the per-context flexibility of scoped behaviour that distinguishes Aurora from other systems.

There is already a large body of work in the area of DSM and DSD systems (some recent examples include [3, 5, 6, 4]). Related work in High Performance Fortran (HPF) and parallel array classes have 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. For example, Munin [3] and Blizzard [13] provide protocols customized to specific data-sharing behaviours. Run-time libraries, such as shared regions [6], 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 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 the parametric shared region mechanism of ABC++ [10]. However, there are two significant differences. First, Aurora allows distributed vectors to be partitioned between different address spaces to improve scalability and to support owner-computes using multiple nodes. A parametric shared region in ABC++ has a single home node, therefore shared data cannot be partitioned. Second, Aurora supports multiple writers to the same distributed vector object, which can be important for performance [4], while parametric shared regions only allow a single writer.

Notably, both ABC++ and scoped behaviour 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, clean-up 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, may 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++) gain all the existing exception handling infrastructure with no extra effort.

7 Concluding Remarks

When developing applications for distributed-memory platforms, such as a network of workstations, shared-data systems are often preferred for their ease-of-use. 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. Aurora achieves good performance through flexible data-sharing policies and by optimizing specific data-sharing patterns. In experiments with a set of parallel programs, we showed that Aurora can achieve comparable performance to MPICH-based programs. What distinguishes Aurora is its use of scoped behaviour to provide per-context and per-object flexibility in applying data-sharing optimizations. We have detailed how scoped behaviour is implemented with a class library and without special compiler support or language extensions to C++. Given the encouraging performance of our current system, we are exploring new scoped behaviours and developing more applications using Aurora.

8 Acknowledgments

Thank you to 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 Ph.D. work at the University of Toronto. Thank you to Toronto's Department of Computer Science and NSERC for financial support. Thank you to ITRC and IBM for their support of the POW Project.

References

[1] G.V. Wilson. *Practical Parallel Programming*. MIT Press, 1995.

[2] P. Lu. Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library. In *Proc. 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 145–158, Portland, Oregon, U.S.A., June 1997. Available at http://www.cs.ualberta.ca/~paullu/.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[10] 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.

[11] 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.

[12] P. Lu. Using Scoped Behaviour to Optimize Data Sharing Idioms. In R. Buyya, editor, *High Performance Cluster Computing: Programming and Applications, Volume 2, 1/e*. Prentice Hall PTR, 1999.

[13] 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.