Chapter 6

Using Scoped Behavior to Optimize Data Sharing Idioms

PAUL LU

Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4
Canada

Email: *paullu@sys.utoronto.ca*

6.1 Introduction

In distributed systems and applications, there are often archetypical reader, writer, and producer-consumer data sharing patterns or idioms. Therefore, in addition to general-purpose data sharing, it is advantageous for distributed systems to flexibly and efficiently support common idioms. Flexibility includes the ability to implement different idioms with minimal and localized changes to the source code. Efficiency includes exploiting the semantics of the idiom to optimize performance.

For example, all message-passing systems provide general-purpose mechanisms to send and receive data across distributed nodes. However, some systems (e.g., Message Passing Interface) also support commonly occurring group operations, such as all-to-all communication. The higher-level functions are implemented using the lower-level mechanisms. By prepackaging a data sharing idiom into a single function, the system facilitates code reuse and it can also exploit the semantics of the idiom to optimize, say, the overlapped sending of different messages.

However, parallel programming systems based on shared memory and shared data models are becoming more popular and widespread. Accessing local and remote data using the same programming interface (i.e., reads and writes) is often more convenient than mixing local accesses with 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) [4], [2] and distributed shared data (DSD) [3], [12], [7] systems. Some hybrid systems combine properties of both DSM and DSD.

At one end of the spectrum, DSM systems typically emulate hardware-based shared memory so that a C-style pointer (e.g., `int *`) can transparently name either local or remote data. DSM systems are usually based on fixed-sized units of sharing, often a page, because they are implemented using the same mechanisms as for demand-paged virtual memory. A single data sharing policy is often 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. Shared data is named and accessed through a programmer's interface and DSD systems usually have a variable granularity of sharing. Consequently, the unit of sharing and the sharing policy can be selected to match the form and function of the particular data.

The Aurora DSD system provides a familiar shared data programming model on distributed memory hardware using standard C++ [9]. All shared data are encapsulated as objects and are accessed through overloaded operators and other interface methods. Using the shared-data objects is designed to be syntactically the same as with built-in data types. Since each shared object is an independent unit of sharing and management, the data access behavior can be optimized on a per-object basis.

Aurora is unique in its use of the scoped behavior approach for applying optimizations (discussed further in Section 6.3.3). With scoped behavior, a new language scope (e.g., nested braces in C++) around selected source code can be used to optimize the data sharing behavior of specific objects. Without any language extensions or special compiler support, the optimizations can be applied on a per-context (i.e., portion of source code) basis. Therefore, the per-object and per-context optimization flexibility allows different loops and computational phases to be optimized in different ways, according to the needs of the application. When used with data placement mechanisms, scoped behavior simplifies the implementation and optimization of a number of common data sharing idioms.

6.2 Motivation: Data Sharing Idioms

In designing computer systems, it is natural to identify and optimize for commonly occurring structures and idioms. Therefore, in addition to general-purpose mechanisms, it is important for the system to easily and seamlessly support higher-level abstractions. For example, in concurrency control, the monitor concept builds upon the underlying lock and semaphore mechanisms.

Analogously, researchers have identified a number of data sharing patterns that reappear in various distributed systems and applications. Data sharing patterns such as read-intensive, write-intensive, producer-consumer, migratory sharing, and others have been noted [4], [6]. Consequently, researchers have explored various compiler and run-time techniques to optimize performance for common data sharing idioms.

Ideally, the distributed system should allow the data sharing behavior of each object to be optimized independently of other objects. False sharing, when performance is degraded by the interference between logically independent data located in the same unit of data management, is a potential problem with DSM systems. As the number of data sharing optimizations increases, the negative impact of interference between sharing policies will likely also increase. In DSD systems, false sharing is avoided by managing independent data as separate objects.

Also, it should be possible to optimize each portion of the source code (i.e., context) independently of other contexts. Therefore, as the data access patterns change from computational phase to phase, it is possible to alter the data sharing policy. In DSD systems, since shared data is an abstract data type, it is natural to consider different programmer interfaces for different contexts.

6.3 Aurora: A Distributed Shared Data System

In this section, we provide a tutorial of the design and mechanisms of the Aurora system.

Aurora is a C++ class library and run-time system that provides a shared data programming model on distributed-memory architectures. Scalar and vector data can be distributed across, and accessed from, different address spaces. To overcome the latency and bandwidth performance problems of typical distributed-memory platforms, Aurora provides a set of data sharing optimizations.

6.3.1 Process Models

Aurora's mixed-parallelism process model includes both task parallelism and data parallelism. Currently, the task parallelism is provided by the underlying ABC++ library through active objects [11]. Active objects are normal C++ objects that contain their own thread of control. Data parallelism is provided by a simple `for_all()` construct that invokes a parallel method on all distributed data.

Aurora also provides a fairly standard single-program, multiple-data (SPMD) process model. The model includes teams of threads, sequential sections (i.e., code to be executed by only one thread, usually the team's "master"), and synchronization barriers.

The various process models can be mixed, in reasonable ways, within a single application.

6.3.2 Shared-Data Objects

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

<table>
<tr><th>(a) Original Source Code</th><th>(b) With Scoped Behavior</th></tr>
<tr><td><pre>GVector<int> vector1(1024);

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

vector1[0] = 1;</pre></td><td><pre>GVector<int> vector1(1024);

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

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

vector1[0] = 1; // Still immediate update</pre></td></tr>
</table>

Figure 6.1 Applying a data sharing optimization using scoped behavior.

the data is on a remote node, since that data access behavior has the least error-prone semantics.

Aurora's data model requires that shared scalar and vector objects be created using Aurora's templated C++ classes `GScalar` and `GVector`:

```
GScalar<int>    scalar1;
GScalar<VNode>  scalar2;          // User-defined type VNode

GVector<int>    vector1( 1024 );  // 1024 elements
GVector<float>  vector2( vsize ); // vsize elements
GVector<VNode>  vector3( vsize ); // User-defined type VNode
```

`scalar1` is a C++ integer that is shared across address spaces. By default, updates to `scalar1` are immediately visible to other processors, which may require a message send for each update. Any of the C++ built-in types or any user-defined concrete type [5], as with `scalar2`, can be an independent unit of sharing.

Shared vectors, such as `vector1`, are similar to shared scalars, except that the vector elements can be distributed across nodes and the subscript operator is overloaded to support indexing. By default, the vector elements are block distributed across all the nodes involved in the computation. Parameters to the constructor specify the size of the vector and (optionally) data placement on the nodes.

Consider the following examples of explicit data placement:

```
// Block distributed across Node 0 and Node 1
GVector<float>  vector4( 1024, Nodes( 0, 1 ) );

// All elements on Node 0
GVector<int>    vector5( vsize, Nodes( 0 ) );
```

Table 6.1 Current Scoped Behaviors

Scoped Behavior	**Comment**
Owner-computes	Threads access only colocated 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.

In the example, `vector4` has 1,024 elements block distributed across nodes 0 and 1, with each block containing exactly 512 elements. `vector5` has `vsize` elements all placed on node 0 (i.e., single block). The class `Nodes` is provided by Aurora and is used during object construction to list the nodes for data placement.

6.3.3 Scoped Behavior

For the system implementor, scoped behavior is a structuring technique to change the implementation of selected methods for the lifetime of a language scope [10]. For the Aurora programmer, scoped behavior is how an optimization is applied to a shared-data object.

For example, if a shared vector is updated in a loop [Figure 6.1(a)] *and* if the updates do not need to be performed immediately, then the loop can use release consistency [1] and batch the writes until the scope is exited [Figure 6.1(b), shown side-by-side for easy comparison]. Without any changes to the loop code itself, the behavior of the updates to `vector1` is changed within the language scope. The `NewBehavior` macro specifies that the release consistency optimization should be *applied* to `vector1` within the loop. Since the statement `vector1[ 0 ] = 1` is outside of the scope, it forms a different context and is not affected by the new behavior.

Therefore, scoped behavior is the main interface between the programming model and the data sharing optimizations (Table 6.1), providing:

- *Per-object* flexibility: The ability to apply an optimization to a specific shared-data object without affecting the behavior of other objects. Within a context, different objects can be optimized in different ways (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.

```
 1:extern int dotProd( int * a, int * b, int j, int n );
 2:
 3:void mmultiply(   // SPMD thread and shared data among parameters
 4:  ProcessTeam * myTeam,
 5:  GVector<int> mA, GVector<int> mB, GVector<int> mC,
 6:  const int m, const int n, const int p )
 7:{
 8:  int i, j;
 9:
10: { // Begin scoped behavior
11:  NewBehavior( mA, GVOwnerComputes, int );
12:  NewBehavior( mB, GVReadCache, int );
13:  NewBehavior( mC, GVReleaseC, int );
14:
15:  while( mA.doParallel( myTeam ) )  // Owner-computes
16:  {
17:    for( i = mA.begin(); i < mA.end(); i += mA.step() )
18:      for( j = 0; j < p; j++ )
19:        mC[i][j] = dotProd( &mA[i][0], mB, j, n );
20:  } // while
21: } // End scoped behavior
22:}
```

Figure 6.2 Matrix multiplication in Aurora, SPMD-style.

6.3.4 Illustrative Example: Matrix Multiplication

Consider the parallel dense matrix multiplication function in Figure 6.2. The Aurora implementation uses the SPMD process model, distributed matrices,[1] and three different data sharing optimizations applied using scoped behavior. At the heart of the function are the nested loops and a C-style dot product function.

Three elements of the SPMD process model are shown in this example. First, the `myTeam` (line 4) parameter points to the local thread executing the function. Each thread has a unique number and each thread is part of a larger team of threads that is executing the same function in the application.

Second, the `NewBehavior` with `mA` (line 11), which uses a macro provided by Aurora, is an example of scoped behavior and owner-computes. Note that this macro exists within a new language scope (i.e., the braces on lines 10 and 21). Outside of the scope, matrix `mA` continues to behave as a object of type `GVector<int>`. Inside the scope, the macro changes the type of matrix `mA` to `GVOwnerComputes<int>`. This changes matrix `mA`'s behavior such that only local data (i.e., the portions of `mA` colocated with, and "owned" by, the thread) can be accessed and it can be ac-

[1]As in C/C++, multidimensional matrices in Aurora are implemented using one-dimensional vectors and a simple indexing function. For clarity, matrix indexing is shown using matrices of static size.

cessed using C-style indexing. Since the scoped behavior is known at compile-time, `NewBehavior` actually changes the code generated by the compiler to use a faster access technique. The user's source code for accessing the shared data is unchanged.

Third, the combination of `doParallel()`, `begin()`, `end()`, and `step()` method calls (lines 15 and 17) guarantees that the computation will span all of the local data. These methods are *not* defined for class `GVector` so they can only be used with the owner-computes scoped behavior. For the local thread in `myTeam`, the `doParallel()` method guarantees to return `true` until all of the local data for the object has been processed. For each call to `doParallel()`, methods `begin()`, `end()`, and `step()` will return the correct parameters to cover a portion of the local data. Using owner-computes for matrix `mA` improves performance by avoiding a large number of remote read accesses and it is a convenient way to partition the parallel work among the different SPMD threads.

However, even after optimizing the accesses to matrix `mA`, performance problems remain because matrices `mB` and `mC` are also distributed. Therefore, the behavior of `mB` within the scope is modified so that each thread makes a local copy of the entire matrix before entering the loops (line 12). Since `mB` is reused many times and since its values do not change, it is cached in local memory for read accesses. Again, since `mB` is of type `GVReadCache<int>` within the scope, the code generated by the compiler is different than without the scoped behavior.

The behavior of `mC` within the scope is modified (line 13) so that all updates to the matrix are not immediately sent, but are buffered until a full network message can be sent or the scope is exited (line 21). Consequently, the cost of sending a message is amortized over many updates to improve performance. This behavior is functionally equivalent to the release consistent memory model for writes. But, in contrast, the synchronization point for remote updates is at the end of the scope and only the updates to the specific data object need to be communicated.

Note that the scoped behaviors of the matrices are all different and are independent of each other (i.e., per-object). They only share the lifetime of the language scope.

6.3.5 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 `NewBehavior` macro. Sometimes, straightforward changes to the looping parameters are needed, as for owners-computes. Consider an alternate way of optimizing the matrix multiplication function: 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. Admittedly, more memory for read caches is required in the alternate 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.

6.4 Implementation Overview

We provide a brief overview of how Aurora's shared-data objects and how scoped behavior are implemented. A more detailed discussion is available elsewhere [10].

6.4.1 Handle-Body Composite Objects

Both shared scalars and vectors in Aurora are implemented through the handle-body idiom for composite objects [5]. The handle class implements the appropriate overloaded operators and type constructors. The handle also provides the location and distribution transparency required to access remote data. In contrast, the body objects contain the actual data. However, to the application programmer, handle and body objects operate as a single object.

Consider a vector of nine elements, say `GVector<int> a( 9 )`, block distributed across three (of four) nodes, as illustrated in Figure 6.3. Assuming block distribution, the bodies of the vector are distributed across three local objects, each (potentially) located in a different address space and on a different physical processor. Handles can be replicated cheaply and passed as parameters, but the local objects are shared among the handles.

Currently, Aurora implements handles as passive objects, but the local objects are implemented as ABC++ active objects. Operations on the member data of an active object are handled by the member functions, as is the case with passive objects. However, unlike passive objects, the member functions of an active object can also be remotely invoked with a smart message. In ABC++, smart messages resemble normal method invocations and automate parameter marshaling and demarshaling. Distributed memory and shared memory ABC++ run-time systems are available to Aurora applications.

By default, a read or write to a shared object is translated into a smart message from the handle to a local body object. Thus, Aurora uses smart messages, in part, for `put()`s to and `get()`s from remote memory.

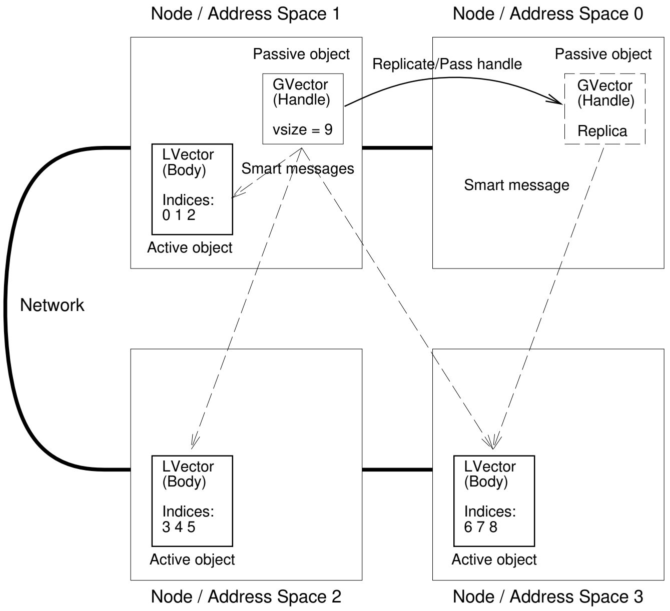


Figure 6.3 Basic handle-body architecture of Aurora.

6.4.2 Scoped Behavior Objects

The main motivation for using language scopes to define the context of scoped behavior is to exploit the property of name hiding. In block-structured languages, an identifier can be reused 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 behavior objects. Due to name hiding, the compiler will generate code according to the class of the scoped behavior object, instead of the original object's class. Aurora also uses the automatic construction and destruction of scoped objects in C++ to properly initiate the required coherence and data movement actions. Therefore, scoped behavior is a combination of compile-time and run-time techniques.

As previously shown in Figure 6.1, programming with scoped behavior under C++ requires a language scope and the `NewBehavior` macro. The macro merely hides the fact that a new handle of type `GVReleaseC<int>` is being created. The macro calls the constructor of class `GVReleaseC` with the old handle to `vector1` as a parameter. Without copying the bodies of `vector1`, the new handle now controls access to the shared vector elements within the scope. The behavior of the new handle is identical to the original handle except that updates are now batched. When the scope is exited, the scoped behavior object is destroyed and the batched updates are flushed by `GVReleaseC`'s destructor.

The new handle is also called `vector1`, thus it lexically hides the old handle, and all references to `vector1` within the new scope are compiled to use the new handle without changes to the user's source code. Consequently, it can be easy to add, change, and experiment with scoped behavior without requiring many error-prone lexical changes to the code.

There are three important aspects to Aurora's mechanisms and architecture:

1. Operator overloading supports location and distribution transparency across address spaces.

2. The handle-body architecture and scoped behavior substitutes handles for the lifetime of a language scope, thus altering the behavior of data accesses.

3. The lexical hiding semantics of nested scopes in C++ (and similar block-structured languages) causes the compiler to generate the appropriate scoped behavior code without changes to the programmer's code.

Although the current set of scoped behaviors is small, the behaviors can be combined on a per-context and per-object basis, and enhanced with explicit data placement, to support a variety of optimization strategies and idioms.

6.5 Experience with Parallel Programs

We discuss the use of scoped behavior in parallel programs and how the data sharing idioms found in these programs are optimized.

The hardware platform is a cluster of workstations assembled as part of the University of Toronto's Parallelism on Workstations (POW) project. Each workstation has a single 133 MHz PowerPC 604 processor and 96 MB of memory. The software includes IBM's AIX 4.1 operating system, POSIX threads (pthreads), the xlC C++ compiler, a prototype version of ABC++, and the MPICH (version 1.1.10) implementation of MPI for interworkstation communication. A single 100 Mbit/s Ethernet switch with half-duplex links is used for these experiments.

The speedups in Figure 6.4 and Table 6.2 are computed against C implementations of the same algorithm (or against quicksort in the case of parallel sorting). In particular, the sequential implementations do not suffer from the overheads of either operator overloading or scoped behavior. Overall, two trends can be noted in the performance results. First, for these three programs, additional processors improve speedup, albeit with diminishing returns. Second, as the size of the data set increases, the overall granularity of work and, thus, speedup also increases.

6.5.1 Matrix Multiplication

Figure 6.4 shows the speedups for a program that uses matrix multiplication. Speedups are given for data sets using 512×512 matrices and using 640×640

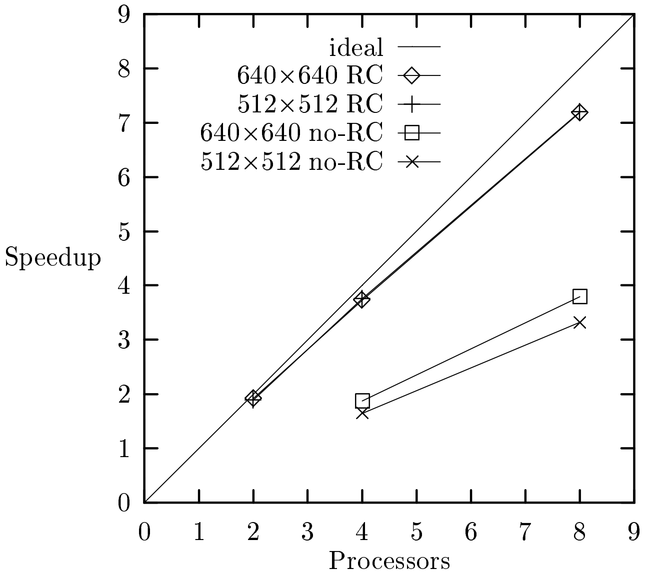


Figure 6.4 Matrix multiplication.

matrices. The real times for eight processors are about 18 seconds and 37 seconds, respectively. The data sets labelled "RC" are when all three scoped behaviors are used for each phase, as in Figure 6.2.

In Phase 1 of the program, `mA` is multiplied with `mB` and assigned to `mC` [i.e., `mmultiply( ..., mA, mB, mC, ... )`]. In Phase 2, `mA` is multiplied with `mC` and assigned to `mB` [i.e., `mmultiply( ..., mA, mC, mB, ... )`]. Of course, the same `mmultiply()` function is used for each phase, but it is called with different parameters. This is a form of per-context flexibility since, in Phase 1, a read cache is used for `mB` and release consistency is applied to `mC`, and vice versa for Phase 2.

Since matrix multiplication is known to be easy to parallelize, the speedups are very good and are virtually identical for the two data sets shown. In terms of data sharing idioms for Phase 1, matrix multiplication is characterized by the read-intensive matrices `mA` and `mB`, and the write-intensive matrix `mC`. In Phase 2, matrix `mB` is write-intensive. As we have seen, read-intensive shared data can be optimized using either owner-computes or a read cache. Write-intensive shared data can be optimized using release consistency. Aurora makes it easy to optimize the different access patterns of the different shared-data objects.

Another benefit of Aurora's scoped behavior approach is that one can easily experiment with and measure the performance impact of different optimization strategies. As a simple example, the program can be executed without the release consistency optimization for matrix `mC` in Phase 1 (and matrix `mB` in Phase

Table 6.2 2-D Diffusion and Parallel Sorting by Regular Sampling

Program	Data Set	Speedup		
		2 PEs	4 PEs	8 PEs
2-D Diffusion	1536 × 1536, 32 time-steps (45.3 seconds on 1 PE)	1.9	3.2	6.6
	1024 × 1024, 32 time-steps (20.2 seconds on 1 PE)	1.9	2.8	5.9
PSRS	10 million keys (58.2 seconds on 1 PE)	n/a	2.5	4.4
	6 million keys (32.8 seconds on 1 PE)	1.3	2.3	4.0

2) by simply commenting out the relevant `NewBehavior` (Figure 6.2, line 13) and re-compiling. No other code changes are needed. Thus, by default, updates to `mC` are sent immediately and results in the performance labelled "no-RC" in Figure 6.4. Since many more update messages are generated, the performance decrease is substantial.

Finally, related to the owner-computes and read cache scoped behaviors, is the ability to bypass the overheads associated with operator overloading, under controlled circumstances. When debugging a program and for noncompute-intensive contexts, the overheads of operator overloading are tolerable. In compute-intensive loops and functions, it is desirable to access data using pointers. In this example, an important contributor to good performance is the use of C-style pointers in `dotProd()`. Note that `dotProd()` does not accept handles to shared-data object (Figure 6.2, line 1). Both the owner-computes and read cache scoped behaviors pass pointers as parameters in the function call to `dotProd()` (Figure 6.2, line 19). In effect, Aurora orchestrates any necessary data movement via scoped behavior and then provides a fast way to access the local data or local read cache through type constructors that return pointers. Not every shared data access needs to be optimized through the use of pointers, but scoped behavior allows the programmer to optimize the program when and where it matters most.

6.5.2 2-D Diffusion

Simulating the diffusion of matter involves data sharing idioms that are archetypical for a class of applications different from matrix multiplication. Therefore, it is important that scoped behavior be able to support idioms beyond the standard read-intensive and write-intensive patterns.

In our simplified simulation of 2-D diffusion, each floating point element of a matrix represents the density of matter at a given coordinate. Two matrices, `timeNow` and `timePrevious`, are used. For each time-step, the new density of matter at a coordinate in `timeNow` is computed by averaging the densities in `timePrevious`

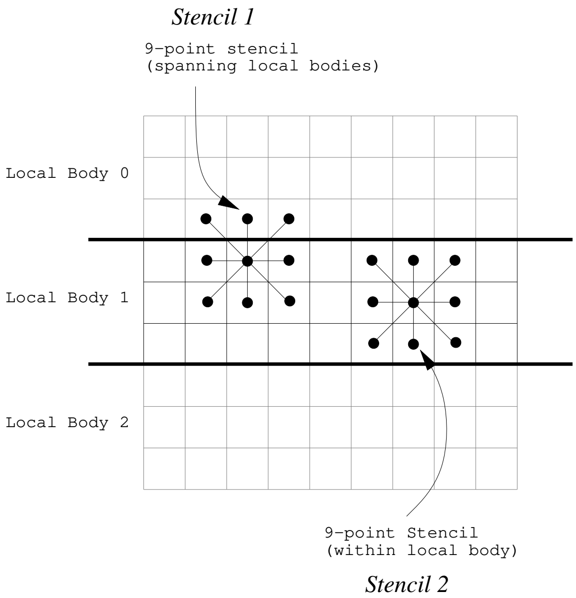


Figure 6.5 Stencils in 2-D diffusion simulation.

at the same coordinate and at its eight nearest neighbors. This computation is known as a 9-point stencil operation (Figure 6.5). For the next time-step, the two matrices are logically swapped.

Assuming a block-distributed matrix, the normal form of owner-computes is a problem because each thread needs to access some matrix elements from a different thread's local body. In particular, at the top and bottom borders of a local body, the nearest neighbors above or below the current coordinate can reside in a different local body. As an example, consider Stencil 1 in Figure 6.5.

Aurora addresses this form of data sharing by supporting block-distributed matrices with border padding and a modified form of owner-computes. These matrices are identical to normal block-distributed matrices except for the extra storage overhead required for the padding. Furthermore, the system automates the movement of the data at the borders into the appropriate border padding of the neighboring local bodies.

In essence, the new scoped behavior establishes a producer-consumer relationship between two neighboring local bodies. All local bodies are producers. The consumers are the neighboring local bodies. For example, Local Body 0 is a producer for Local Body 1, since the bottom row of Local Body 0 is used as part of computing Stencil 1 for Local Body 1. The data that is produced is moved into the border padding storage of the consumer. Once the automatic data movement is

Table 6.3 Per-context and Per-object Optimizations in PSRS

Phase	Algorithm	Main Optimizations
1	Sort local data and gather samples.	Sort local data using owner-computes. Samples are colocated with master and are gathered using release consistency.
2	Sort samples, select pivots, and partition local data.	Master sorts samples using owner-computes. Pivots are accessed from a read cache.
3	Gather partitions.	Partitions are gathered into a read cache using `distmemcpy()`.
4	Merge partitions and update local data.	Merge partitions from the read cache.

complete, the computationally-intensive 9-point stencil operation can be efficiently performed using a C-style pointers on the local body plus padding.

A total of 32 time-steps are simulated on a 1024×1024 grid and on a 1536×1536 grid, with the performance given in Table 6.2. Since a barrier synchronization is required after each time-step in the current implementation, the granularity of work is low. A larger data set and granularity of work would help improve the speedups for this parallel program.

Of course, the padding strategy used by Aurora is not new. However, Aurora abstracts the implementation details of the padding and automates the data movement using scoped behavior. Also, in keeping with the abstract data type approach, the new scoped behavior can be reused in similar applications.

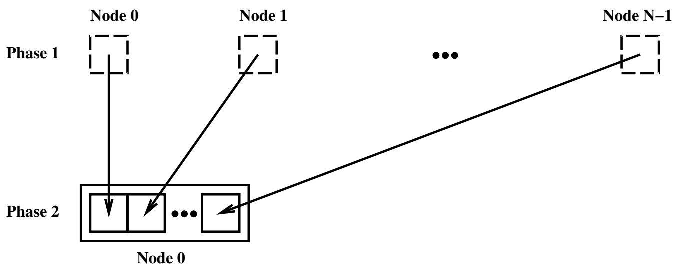
6.5.3 Parallel Sorting by Regular Sampling

The Parallel Sorting by Regular Sampling (PSRS) application kernel [13], [8] has been implemented using Aurora. The performance of PSRS is given in Table 6.2. PSRS is a general-purpose, comparison-based sort with key exchange. It can be communication-intensive since the number of keys exchanged grows linearly with the problem size.

The basic PSRS algorithm consists of four distinct phases. Assume that there are p processors and the original vector is block distributed across the processors. Phase 1 does a local sort (usually via quicksort) of the local block of data. No interprocessor communication is required for the local sort. Then, a small sample (usually p) of the keys in each sorted local block is gathered by each processor. In Phase 2, the gathered samples are sorted by the master process and $p-1$ pivots are selected. These pivots are used to divide each of the local blocks into p partitions.

```
  GVector<int> Sample( vsize, Nodes( 0 ) );

  // Phase 1:  All nodes are producers
{
  NewBehavior( Sample, GVReleaseC, int );
  for( i = ... )
      Sample[ i ] = ...              // Produce
}
```




```
  // Phase 2:  Only Node 0 (i.e., master) is the consumer
MASTER
{
  NewBehavior( Sample, GVOwnerComputes, int );
  quicksort( Sample, 0, vsize - 1 );  // Consume
}
```

Figure 6.6 Producer-consumer idiom in PSRS.

In Phase 3, each processor i keeps the ith partition for itself and gathers the ith partition of every other processor. At this point, the keys owned by each processor fall between two pivots and are disjoint with the keys owned by the other processors. In Phase 4, each processor merges all of its partitions to form a sorted list. Finally, any keys that do not reside in the local data block are sent to their respective processors. The end result is a block-distributed vector of sorted keys.

A multiphase algorithm with several shared-data objects, like PSRS, is particularly suited for the per-context and per-object data sharing optimizations in Aurora. The optimizations required, and the objects that are optimized, differ from one phase to another phase. Table 6.3 summarizes the phases of PSRS and the main Aurora optimizations. Note that the data movement in Phase 3 is implemented with a `memcpy()`-like construct (called `distmemcpy()`) with full location and distribution transparency.

The programmer's knowledge of the application's data sharing idioms can also be exploited. For example, there is a multiple-producer and single-consumer sharing pattern during the gathering of sample keys in Phases 1 and 2 (Figure 6.6). Note that the vector `Sample` has been explicitly placed on Node 0. In Phase 1, the samples are gathered by all nodes and optimized using release consistency. Each node updates a disjoint portion of the vector. In Phase 2, the master node (i.e., Node 0) sorts all of the gathered samples. Since `Sample` is colocated with the master, we can use the owner-computes optimization and call `quicksort()` using a C-style pointer to the local data, for maximum performance.

Two other implicit optimizations are also present. First, `Sample` is updated in Phase 1 without unnecessary data movement and protocol overheads. With many page-based DSM systems, updating data requires the writer to first demand-in and gain exclusive ownership of the target page. However, with Aurora, the system does not need to demand-in the most current data values for `Sample` because it is only updated and not read in Phase 1. And, since the different nodes are updating disjoint portions of the vector, there is no need to arbitrate for ownership of the page to prevent race conditions. By design, the scoped behavior allows the programmer to optimize disjoint, update-only data sharing idioms involving multiple writers.

Second, since the local body for `Sample` is explicitly placed on Node 0, the updated values are sent eagerly and directly to the master node when the Phase 1 scope is exited. Therefore, there is no need to query for and demand-in the latest data during Phase 2, as would be the case for many DSM systems.

6.6 Discussion and Related Work

A large body of work has emerged in the area of DSM and DSD systems. These systems vary in their strategies to overcome false sharing, tolerance for high communication costs [4], [2], reliance on compiler support [3] and implementation techniques [12], [7], and their extensibility. Related work in High Performance Fortran (HPF) and parallel array classes have also addressed the basic problem of transparently sharing data.

One refinement in Aurora's implementation of release consistency is that only the affected data objects are made consistent at the end of a scope, instead of an entire page or all of memory. We refer to this as per-object release consistency. In this way, Aurora partitions the consistency namespace to avoid unnecessary updates of shared data.

The basic ideas behind scoped behavior are not new. The notion of nested scopes is fundamental to block-structured sequential languages. The association of actions with C++ constructors and destructors is also not new. For example, ABC++ also associates data movement actions with the scoped lock handles of its parametric shared regions [11]. Aurora's innovation is the application of scoped behavior, with the handle-body architecture, to the problem of per-context and per-object data sharing optimization.

In a largely C context, shared regions [12] and CRL [7], associate run-time coherence actions with programmer-inserted code annotations (i.e., function calls). One advantage of the automatic invocation of constructors and destructors in Aurora is that it is impossible to omit an annotation and miss a coherence action. Aurora is also able to exploit compile-time information to generate different code for different behavior.

More generally, scoped behavior, as an interface and mechanism, is similar to compiler directives and annotations. However, as an alternate implementation technique, scoped behavior does not require complex compiler support and its class library approach is easier to extend than modifications to a compiler [10].

6.7 Concluding Remarks

We have described and explored an all-software, standard C++-based DSD system called Aurora. The system is unique in its use of scoped behavior for data sharing optimizations. The advantages of Aurora's approach include its ability to support a variety of reader, writer, and producer-consumer data sharing idioms.

We have experimented with Aurora using three parallel programs and identified some of the data sharing idioms found in the applications. Matrix multiplication illustrates the effectiveness of Aurora at optimizing read-intensive and write-intensive access patterns. We also explored how a variety of optimization strategies can be tried with minimal error-prone source code changes. The 2-D diffusion program showed how special-purpose data movement idioms can be supported using scoped behavior. And the parallel sorting application demonstrates the value of per-context optimization for multiphase algorithms.

In distributed systems and applications, it is natural to identify and optimize for commonly occurring structures and idioms. Aurora's ability to combine scoped behaviors on a per-context and per-object basis, and to control data placement, allows important data sharing idioms to be optimized.

Acknowledgments

Thank you to Ben Gamsa, Kate Keahey, Bill O'Farrell, Eric Parsons, Karen Reid, John Reynders, Jonathan Schaeffer, Ken Sevcik, Michael Stumm, Roel van der Goot, Greg Wilson, and Songnian Zhou for their 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.

6.8 Bibliography

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

[2] 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*, vol. 29(2), pages 18–28, February 1996.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 18(3), pages 190–205, March 1992.

[4] 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*, Seattle, WA, pages 168–176, March 1990.

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

[6] 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*, Washington, DC, pages 380–389, November 1994.

[7] 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*, Copper Mountain Resort, CO, pages 213–228, December 1995.

[8] 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*, vol. 19, pages 1079–1103, October 1993.

[9] P. Lu. Aurora: Scoped Behavior for Per-Context Optimized Distributed Data Sharing. In *Proc. 11th International Parallel Processing Symposium*, Geneva, Switzerland, pages 467–473, April 1997.

[10] P. Lu. Implementing Optimized Distributed Data Sharing Using Scoped Behavior and a Class Library. In *Proc. 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR, pages 145–158, June 1997.

[11] 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, Cambridge, MA, 1996.

[12] 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*, San Diego, CA, pages 229–238, May 1993.

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