Journal of Parallel and Distributed Computing **61**, 1609–1632 (2001)
doi:10.1006/jpdc.2001.1758, available online at http://www.idealibrary.com on IDEAL®

Integrating Bulk-Data Transfer into the Aurora Distributed Shared Data System

Paul Lu

Department of Computing Science, University of Alberta, Edmonton, Alberta T6G 2E8, Canada
E-mail: paullu@cs.ualberta.ca

Received January 1, 2000; revised August 19, 2000; accepted October 11, 2000

The Aurora distributed shared data system implements a shared-data abstraction on distributed-memory platforms, such as clusters, using abstract data types. Aurora programs are written in C++ and instantiate shared-data objects whose data-sharing behaviour can be optimized using a novel technique called scoped behaviour. Each object and each phase of the computation (i.e., use-context) can be independently optimized with per-object and per-context flexibility. Within the scoped behaviour framework, optimizations such as bulk-data transfer can be implemented and made available to the application programmer. Scoped behaviour carries semantic information regarding the specific data-sharing pattern through various layers of software. We describe how the optimizations are integrated from the uppermost application-programmer layers down to the lowest UDP-based layers of the Aurora system. A bulk-data transfer network protocol bypasses some bottlenecks associated with TCP/IP and achieves higher performance on an ATM network than either TreadMarks (distributed shared memory) or MPICH (message passing) for matrix multiplication and parallel sorting.

Key Words: bulk-data transfer; distributed-memory computing; shared data; data-sharing patterns; optimizations; scoped behaviour; network of workstations; clusters.

1. INTRODUCTION

Distributed-memory platforms, such as networks of workstations and clusters, 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. Also, existing network protocols, such as TCP/IP, were not originally designed for communication-intensive clusters and may not be the best choice for performance.

Systems based on shared-memory and shared-data models are becoming increasingly popular for distributed applications. Broadly speaking, there are *distributed shared memory* (DSM) [1, 4] and *distributed shared data* (DSD) [2, 14] 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.

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 or methods implement the data-sharing policy. If an object-oriented language is used, the ADT can be a shared-data object.

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. 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 are 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.

TABLE 1

Layered View of Aurora

Layer	**Main components and functionality**
Programmer's interface	Process models (data parallelism, task parallelism, threads, active objects) Distributed vector and scalar objects *Scoped behaviour*
Shared-data class library [10]	Handle-body shared-data objects Scoped handles implementing data-sharing optimizations
Run-time system	Active objects and remote method invocation (currently, ABC++ [12]) Threads (currently, POSIX threads) Communication mechanisms (shared memory, MPI, UDP sockets)

Flexibility is important because different loops or phases of a computation can have different access patterns for the same data structure. For example, in a chain or pipeline of computational phases, the output of one phase becomes the input of the next phase. A data structure may have a read-only access pattern during one phase and a read-modify-write access pattern during a later phase. If the same data structure is large and has to be communicated to all processes, a third data-sharing policy may be required to efficiently handle the bulk-data transfer. Therefore, it is desirable to be able to select a different data-sharing policy for each computational phase.

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. Once the programmer has selected an optimization, the high-level semantics of the optimization are also carried across various software (and perhaps hardware) layers. For example, in an all-to-all data exchange, knowledge about the senders, the receivers, and the size and layout of the data to be communicated, can be important in the implementation of the optimization.

We begin by illustrating and summarizing Aurora's programming model. Then we illustrate how two multi-phase applications, a two-stage matrix multiplication and parallel sorting, require a bulk-data transfer and can be optimized using scoped behaviour. Finally, we demonstrate how the Aurora programs can outperform comparable implementations using a DSM system and a message-passing system on a cluster of workstations with an ATM network.

2. SCOPED BEHAVIOUR AND THE AURORA SYSTEM

Aurora is a parallel programming system that provides novel abstractions and mechanisms to simplify the task of implementing and optimizing parallel programs. Aurora is an object-oriented and layered system (Table 1) that uses objects to abstract both the processes and shared data to provide a complete programming environment.

The single most novel aspect of Aurora is the *scoped behaviour* abstraction. Scoped behaviour is an application programmer's interface (API) to a set of system-provided optimizations; it is also an implementation framework for the optimizations. Unlike typical APIs based on function calls, scoped behaviour integrates the design and implementation of all the software layers in Aurora using both compile-time and run-time information. Consequently, there are a number of ways to view scoped behaviour, depending on the specific software layer.

1. To the application programmer, scoped behaviour is the interface to a set of pre-packaged data-sharing optimizations that are provided by the Aurora system. Conceptually, scoped behaviour is similar to compiler annotations. The application programmer uses Aurora's classes to create shared-data objects. Then, scoped

behaviour is used to incrementally optimize the data-sharing behaviour of a parallel program with a high degree of flexibility. In particular, scoped behaviour provides:

- *Per-context flexibility*. The ability to apply an optimization to a specific portion of the source code. A language scope (i.e., nested braces in C++) around source code defines the context of an optimization. Different portions of the source code (e.g., different loops and phases) can be optimized in different ways.
- *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).

By combining both the per-context and per-object flexibility aspects of scoped behaviour, the application programmer can optimize a large number of data-sharing patterns [9].

2. To the implementor of the class library, scoped behaviour is how a variety of data-sharing optimizations can be implemented by temporarily changing an object's interface [10]. Scoped behaviour does not require language extensions or special compiler support, thus it requires less engineering effort to implement than new language constructs or compiler annotations. As an implementation framework, scoped behaviour can exploit both compile-time and run-time information about the parallel program.

3. To the implementor of the run-time system, scoped behaviour is a mechanism for specifying high-level semantic information about how shared data are used by the parallel program. Scoped behaviour can also carry semantic information about the senders, the receivers, and the specific data to be communicated, across layers of software.

2.1. Example: A Simple Loop

Aurora supports both shared scalars and shared vectors. A scalar object is placed on a specific *home node*, but the shared data within the object can be accessed from any processor node. In contrast, a distributed vector object can have a number of different home nodes, with each processor node containing a different portion of the shared data. The shared vector elements are distributed among different processor nodes, but the data can be accessed from any processor node, as with scalar objects. Currently, only block distribution is supported for shared vectors. The shared data can be replicated and cached, but the home node(s) of the data does not migrate.

Once created, a shared-data object is accessed transparently using normal C++ syntax, regardless of the physical location of the data. Overloaded operators, and other methods in the class, translate the data accesses into the appropriate loads, stores, or network messages depending on whether the data are local or remote. Therefore, as with DSM systems, Aurora provides the illusion that local

(a) Original Loop

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



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

(b) Optimized Loop Using Scoped Behaviour

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

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

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

FIG. 1. Applying a data-sharing optimization using scoped behaviour.

and remote data are accessed using the same mechanisms and syntax. In reality, Aurora uses an ADT to create a shared-data abstraction.

Aurora's data model requires that shared scalar and vector objects be created using Aurora's C++ class templates `GScalar` and `GVector`. Any of the C++ built-in types or any user-defined concrete type [5] can be an independent unit of sharing.

Figure 1a demonstrates how a distributed vector object is instantiated and accessed. Note that `vector1` is a shared 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 method that determines whether the update to `vector1` at index `i` is local or remote. If the data are local, a write is simply a store to local memory. If the data are 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 are read from and written to synchronously, even if the data are on a remote node, since that data-access behaviour has the least error-prone semantics.

The implementation details have been discussed elsewhere [10], but we briefly sketch the main ideas at this time to provide intuition about the implementation and motivate the need for data-sharing optimizations. Since Aurora has defined class `GVector` such that the subscript operator is overloaded, the syntax `vector1[i]` is equivalent to `vector1.operator[](i)`, where `operator[]()` is the name of a class method and vector index `i` is a parameter to the method. When `vector1[i]` is assigned a value, the subscript operator method looks up the processor node on which index `i` is located and the element at that index is updated. The internal data structures of `vector1` keep track of, and can locate, where all the vector elements are stored. Since `vector1` is block distributed across all the processor nodes, some of the vector elements are on the same node as the thread that is executing the update loop. The updates to these co-located vector elements are translated into a simple store to local memory. For all the other vector elements which are not co-located with the thread, the updates are translated to a network message to the remote processor node. A thread in the Aurora run-time system on the remote node reads the message and performs the actual update in its local memory. Then, the thread on the remote node sends a network message back to the thread executing the loop, which has been blocked and waiting for the acknowledgment. This is the well-known request-response message pattern.

Since the default policy for writing to shared-data objects, such as `vector1`, is synchronous updates, the thread executing the update loop must wait for the acknowledgment message from the remote node. In doing so, the update to vector index `i` is guaranteed to be completed before the update to index `i+1` of the loop. Theoretically, if a second thread is reading index `i` of `vector1` at the same time that index `i+1` is being updated, then the second thread should read the value that was just stored at index `i`. Conceptually, synchronous updates are what programmers are familiar with in sequential programs, which is why it was chosen as the default policy.

However, synchronous updates are slow. Two network messages, an update and an acknowledgment, and various context switches are required to update vector elements on a remote node. For typical distributed-memory platforms, two messages can take thousands of processor cycles. If the semantics of the application require synchronous updates, then little can be done to improve performance. However, if the programmer knows that synchronous updates are not necessary for correctness, then the write-intensive data-sharing pattern of the loop can be optimized.

Consider the case where the shared vector is updated in a loop, but the updates do not need to be performed synchronously. For example, the application programmer may know that no other thread will be reading from the vector until after the loop. In such a case, the programmer can choose to buffer the writes, flush the buffers at the end of the loop, and batch-update the shared vector (Fig. 1b). So, instead of two network messages for each update to a remote node, multiple updates are sent in a single network message and there is one acknowledgment for the whole buffer. If a buffer holds hundreds of updates, then the performance improvement through amortizing the overheads is substantial.

Three new elements are required to use scoped behaviour to specify the optimization (Fig. 1b): opening and closing braces for the language scope and a system-provided macro. Of course, the new language scope is nested within the original scope and the new scope provides a convenient way to specify the context of the optimization.

The `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.

At the shared-data class library layer (Table 1), the optimization is implemented by creating a new handle (or wrapper) around the original `vector1` object [8, 10]. The new handle object only exists within the nested scope and the object is of class `GVReleaseC`. By redefining the access methods for `GVector` inside `GVReleaseC` (i.e., compile-time), the optimization is implemented. By creating a new object within the nested language scope, the application source code that uses `vector1` does not have to be modified since, by the rules of nested scopes in block-structured languages, the redefined code for the new handle will be automatically selected. Actions, such as creating and flushing buffers, can be dynamically associated

(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

int mA[ 512 ][ 512 ];
int mB[ 512 ][ 512 ];
int mC[ 512 ][ 512 ];

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

GVector<int> mA( MatrixDim( 512, 512 ) );
GVector<int> mB( MatrixDim( 512, 512 ) );
GVector<int> mC( MatrixDim( 512, 512 ) );

{ // 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
```

FIG. 2. Matrix multiplication in Aurora.

with the constructor and destructor of the handle object inside the new scope (i.e., run-time) in a create-use-destroy model of the handles to the shared-data objects.

2.2. *Example: Matrix Multiplication*

We now consider a more complex example involving multiple shared-data objects and different scoped behaviour optimizations, namely that of nonblocked, dense matrix multiplication, as shown in Fig. 2. The basic 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 (Fig. 2a). The basic algorithm consists of three nested loops, where the innermost loop computes a dot product and can be factored into a separate C-style function.

However, each matrix has a different access pattern and different properties (Fig. 3). Different scoped behaviour optimizations can be applied to different shared-data objects. In particular:

1. Matrix A is read-only. Also, each row is independent of other rows in that it is never necessary to read multiple rows of Matrix A when computing a given row in Matrix C.

2. Matrix B is read-only. Also, since all of Matrix B is accessed for each row of Matrix A, the working set for Matrix B is large.

Given the size of Matrix B, it may be most efficient to move this data using bulk-data transfer.

```
// Sequential matrix multiplication
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    mC[i][j] = dotProd( &mA[i][0], mB, j, size );
```

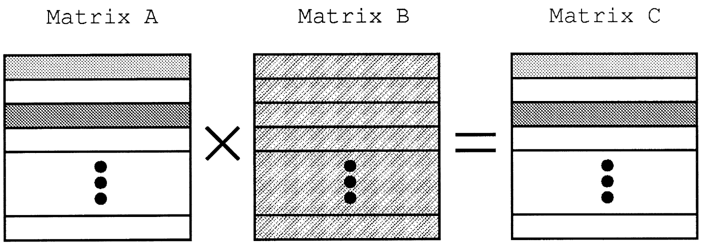

FIG. 3. Matrix multiplication: Different access patterns.

3. Matrix C is write-only. Specifically, the updated values in Matrix C do not depend on the previous values in Matrix C. During the multiplication itself, the previous values of Matrix C are never read.

Conceptually, we can view an optimization as a change in the type of the shared object for the lifetime of the scope. As already discussed, the actual implementation is based on the dynamic creation and destruction of nested handle objects with redefined access methods in their classes (i.e., the new type for the original shared-data object). As an example of per-object flexibility, three different data-sharing optimizations (Table 2) are applied to the sequential code in Fig. 2b to create the parallel code in Fig. 2c.

We describe the scoped behaviours in increasing order of complexity. Since the scoped behaviours for vectors `mC` and `mB` require no changes to the source, we describe them first. The scoped behaviour for vector `mA` is more complicated and it requires some modest changes to the source code, so we discuss it last.

The scoped behaviours are:

1. `NewBehaviour(mC, GVReleaseC, int)`. To reduce the number of update messages to elements of distributed vector `mC` during the computation, the type of `mC` is changed to `GVReleaseC`. As with the simple loop example, the overloaded subscript operator batches the updates into buffers and messages are only sent when the buffer is full or when the scope is exited. Also, multiple writers to the same distributed vector are allowed. No lexical changes to the source code are required.

2. `NewBehaviour(mB, GVReadCache, int)`. To automatically create a local copy of the entire distributed vector `mB` at the start of the scope, the type of `mB` is changed to `GVReadCache`. Caching vector `mB` is an effective optimization because the vector is read-only and reused many times. The challenge, as will be discussed later, lies in how to efficiently transfer the required data into all of the read caches. At the end of the scope, the cache is freed.

Note that `dotProd()` expects C-style pointers (i.e., `int *`) as formal parameters `a` and `b`. Pointers provide the maximum performance when accessing the contents of vector `mB`. Therefore the read cache scoped behaviour includes the ability to pass a

TABLE 2

Example Scoped Behaviours

Scoped behaviour	Description
Owner-computes	Threads access only co-located data.
Caching for reads	Create local copy of data.
Release consistency	Buffer write accesses.
Combined "caching for reads" and "release consistency"	Read from local cache *and* buffer write accesses.
Read-mostly	Read from local copy; eager updates to replicas on write. Currently only implemented for scalars. Good for read-only and read-mostly variables.

C-style pointer to the newly created cache as the actual parameter to `dotProd()`'s formal parameter `b`. Note that no lexical changes to the loop's source code are required for this optimization.

3. `NewBehaviour(mA, GVOwnerComputes, int)`. To partition the parallel work, the owner-computes technique is applied to distributed vector `mA`.

Owner-computes specifies that only the thread co-located with a data structure, or part of a data structure, is allowed to access the data. These data accesses are all local accesses. Given a block-distributed vector, the different threads of an SPMD team of threads are co-located with different portions of the vector. Thus, each of the threads in the team will access a different portion of the distributed vector.

Within the scope, vector `mA` is an object of type `GVOwnerComputes` and has special methods `doParallel()`, `begin()`, `end()`, and `step()`. Only the threads that are co-located with a portion of `mA`'s block-distributed data actually enter the `while` loop and iterate over their local data. It is possible that some processes are located on nodes that do not contain a portion of the partitioned and distributed vector `mA`. These processes do not participate in the computation because they do not enter the body of the `while` loop.

Note that function `dotProd()` also expects a pointer for formal parameter `a`. Since the portions of vector `mA` to be accessed are in local memory, as per owner-computes, it is possible to use a pointer. Therefore, `GVOwnerComputes` provides a C-style pointer to the local data as the actual parameter to `dotProd()`'s parameter `a`. Although some changes to the user's application source code are required to apply owner-computes, they are relatively straightforward.

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. That is to say, the locality of data references is greatly improved. In addition, the parallel program uses the same efficient, pointer-based `dotProd()` function as in the sequential program.

Furthermore, the high-level semantics of scoped behaviours can be exploited for further efficiencies [9]. Typical demand-paged DSM systems do not exploit knowledge about how the data are accessed. For example, even when each element of a data structure is eventually accessed (i.e., dense accesses), DSM systems send an individual request message for each page of remote data. However, scoped behaviours do contain extra semantic information. The read cache scoped behaviour specifies that *all* of vector `mB` is cached, therefore there is no need to transfer each unit of data separately. The multiple request messages can be eliminated if the data are streamed into each read cache via bulk-data transfer. Although the notion of a bulk-data protocol is not new, scoped behaviour provides a convenient implementation framework to exploit the high-level semantics. As another example, vector `mC` is write-only, as opposed to read-write; therefore we can avoid the overhead of demanding in the data since it is never read before it is overwritten. Without *a priori* knowledge that the data is write-only, a programming system must assume the most general and most expensive case of read-write data accesses. Scoped behaviour can capture *a priori* knowledge about how data are used.

3. PERFORMANCE EVALUATION

We now compare and contrast the performance of two applications implemented using three different types of parallel programming systems: Aurora, TreadMarks (a page-based DSM system), and MPICH (a message-passing system). A subset of the performance results with Aurora and MPICH has been previously reported [10]. The results with TreadMarks are new and a larger cluster (16 nodes instead of 8 nodes) has been used for this performance evaluation.

Although there are differences in the implementations of the programs using the different systems, care has been taken to ensure that the algorithms and the purely sequential portions of the source code are identical. Also, different datasets for each application are used to broaden the analysis and to highlight performance trends.

3.1. Experimental Platform

The hardware platform used for these experiments is a 16-node cluster of IBM RISC System/6000 Model 43P workstations, each with a 133 MHz PowerPC 604 CPU, at least 96 MB of main memory, and a 155 Mbit/s ATM network with a single switch. The ATM network interface cards are FORE Systems PCA-200 EUX/OC3SC, which connect to the PCI bus of the workstation. The ATM switch is a FORE Systems Model ASX-200WG. This cluster was assembled as part of the University of Toronto's Parallelism on Workstations (POW) project.

The software includes IBM's AIX operating system (version 4.1), AIX's built-in POSIX threads (Pthreads), the `xlC_r` C/C++ compiler (version 3.01), and the ABC++ class library (version 2, obtained directly from the developers in 1995). TreadMarks (version 0.10.1) is used as an example DSM system.

The run-time system of ABC++, which is also part of the Aurora run-time system (Table 1), uses the MPICH (version 1.1.10) implementation of the Message-

Passing Interface (MPI) as the lowest user-level software layer for communication [6]. For our platform, MPICH uses sockets and TCP/IP for data communication. The ABC++ run-time system is a software layer above MPICH and adds threads to the basic message-passing functions. A daemon thread regularly polls for and responds to signals generated by incoming MPICH messages. The daemon can read messages concurrently with other threads that send messages. Although the daemon incurs context switching and polling overheads, it also has the benefit of being able to pull data off the network stack in a timely and automatic manner.

The message-passing programs also use MPICH, but without the multiple threads used in the ABC++ run-time system. These message-passing programs are linked with the same MPICH libraries as in the Aurora system. Of course, there are several different implementations of the MPI standard, each with their performance strengths and weaknesses. Therefore, for precision, we will refer to these message-passing programs as MPICH programs for the rest of this discussion.

All programs, Aurora, the ABC++ class library, MPICH, and TreadMarks are compiled with `-O` optimization.

3.2. *Applications, Datasets, and Methodology*

As summarized in Table 3, the applications are a matrix multiplication program (MM2) and a parallel sort via the Parallel Sorting by Regular Sampling (PSRS) algorithm [7]. We have also experimented with a 2-D diffusion simulation and the travelling salesperson (TSP) problem [11], but those applications do not have a bulk-data transfer component so we do not discuss them here.

Matrix multiplication is a commonly used application from the literature and we have extended it by using the output of one multiplication as the input of another multiplication. In practice, the output of one computation is often used as the input of another computation. Parallel sorting is an application that has been widely studied by researchers because it has many practical uses. For this study, we are primarily interested in the data-sharing behaviour of the applications, as noted in the comments section of Table 3.

By design, large portions of source code are identical in the TreadMarks, Aurora, and MPICH implementations of the same application. In this way, differences in performance can be more directly attributed to differences in the programming systems.

The speedups of all the programs are computed against sequential C implementations of the same algorithm (Table 3). In the case of PSRS, quicksort is used for the sequential times. Therefore, the typical object-oriented overheads (e.g., temporary objects and scoping) are not part of the sequential implementations, but *are* part of the parallel implementations.

Unless noted otherwise, the reported real times are the averages of five runs executed within a single parallel job. More specifically, the processes (or a single process) are started up, the data structures are initialized, the data pages are touched to warm up the operating system's page tables, and then the computational core of the application is executed five times within an outer loop. Measurement

TABLE 3

Summary of Applications and Datasets

Application	Dataset	Time (seconds)		Comments
Matrix multiplication (MM2)	512×512 matrices	110.6	(1 PE)	Original implementation was for Aurora. Computes $P \leftarrow Q \times R$ then $R \leftarrow Q \times P$. Initial values are
		11.7	(16 PE, MPICH)	
			Speedup of 9.49	
	704×704 matrices	250.7	(1 PE)	randomly generated integers. Allgather data-sharing pattern (Figure 4.1 of [16]).
		25.2	(16 PE, MPICH)	
			Speedup of 9.94	
Parallel sorting by regular sampling (PSRS)	6 million keys	14.4	(1 PE)	Original implementation for shared memory [7]. Keys are randomly generated 32-bit integers.
		3.68	(16 PE, MPICH)	
			Speedup of 3.91	
	8 millions keys	19.9	(1 PE)	Multi-phase algorithm. Broadcast, gather, and all-to-all patterns (Figure 4.1 of [16]).
		11.21	(16 PE, MPICH)	
			Speedup of 1.78	

error, the activity of other processes on the system (e.g., daemon processes, disk I/O), and other factors can cause small variations in the real times. Therefore, each run is timed and the average time of the five runs is taken to be the solution time. Unless noted otherwise, the observed range (i.e., minimum to maximum) of real times of the runs is low relative to the total run time.

The datasets for matrix multiplication and PSRS are randomly generated. A different random number seed is used for each run. For example, in matrix multiplication, the matrices contain values that are randomly generated using a different seed for each run. Similarly, the keys sorted by PSRS are uniformly distributed 32-bit integers that are randomly generated with a different seed value for each run. Different random number seeds are used to help eliminate anomalies in the initial random ordering of keys for a given dataset size.

Throughout this discussion, the MPICH times are used as the baseline benchmark since it is generally acknowledged that message-passing programs set a high standard of performance. Even though the MPICH programs are not always the fastest, as can be seen in Table 3, they define the baseline in order to be consistent.

3.3. Matrix Multiplication

Design and Implementation. The matrix multiplication application used for this evaluation is different from the program discussed in Section 2.2 in that two separate matrix multiplications are performed in succession. There are two phases separated by a barrier. In Phase 1, $P \leftarrow Q \times R$ is computed. In Phase 2, $R \leftarrow Q \times P$ is computed. Note that matrix P is written to in the first phase and it is read from in the second phase. Thus, the output of one multiplication is used as the input of the next multiplication and the optimization needs of the matrices change from phase-to-phase. It is assumed that all three matrices are block distributed across the processors in the parallel job. Although the specific matrix computation is synthetic, it is designed to reflect how shared data are used in real applications.

In Phase 1, matrices Q and R are read-intensive and matrix P is write-intensive. In Phase 2, matrices Q (again) and P are read-intensive and matrix R is write-intensive. As previously discussed, 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. Alternatively, write-intensive accesses can also be optimized using owner-computes if the data are appropriately distributed. Since the access patterns for matrices P and R change from phase to phase, the per-context flexibility of scoped behaviour is particularly valuable.

In the Aurora implementation, the same matrix multiplication function `mmultiply()` is used for both phases, but the function is called with different shared-data objects as the actual parameters. Function `mmultiply()` has formal parameters `mA`, `mB`, and `mC`, which are all `GVectors`, and the function always computes `mC` $\leftarrow$ `mA` $\times$ `mB`. In contrast to Fig. 2, the owner-computes scoped behaviour is applied to both `mA` and `mC` and the read cache scoped behaviour is applied to `mB`. We can use owner-computes for `mC` because it is block distributed. In Phase 1 of the program, `mQ` is multiplied with `mR` and assigned to `mP` [i.e., the program calls `mmultiply(..., mQ, mR, mP, ...)`]. In Phase 2, `mQ` is multiplied with `mP` and assigned to `mR` [i.e., the program calls `mmultiply(..., mQ, mP, mR, ...)`]. Calling function `mmultiply()` with different actual parameters is one form of per-context flexibility since data-sharing optimizations will be applied to different matrices, depending on the call site.

Note that the only data sharing is for the read cache of formal parameter `mB`. Since matrix `mB` is block distributed, loading the read cache always results in an all-to-all communication pattern as each node sends a copy of its local portion to all other nodes, and reads the data from the other nodes into a local cache. Snir *et al.* describe this specific data-sharing pattern as "allgather" [16, Fig. 4.1]. Furthermore, each node sends the exact same data to each of the other nodes, which is different from the all-to-all communication pattern in the PSRS application discussed below. The pattern in PSRS is described as "alltoall (vector variant)" [16].

In TreadMarks, the matrices are allocated from the pool of shared pages, so each page of the shared matrix `mB` is demanded-in as it is touched by the nested loops of the multiplication. Both the TreadMarks and MPICH programs have the exact same `mmultiply()` function, with the matrix parameters passed as C-style pointers. After the entire matrix `mB` has been locally cached, there is no more data communication. As with Aurora, matrix `mB` is actually matrix `mR` during Phase 1. In Phase 2, matrix `mP` is the actual parameter and thus must be demanded-in during that phase. Since matrix `mP` is updated in Phase 1, reading from matrix `mP` in Phase 2 invokes the relevant data consistency protocols in TreadMarks. There are no per-matrix or per-context optimizations in TreadMarks.

For MPICH, the actual data for formal parameter `mB` is explicitly transferred into a buffer before `mmultiply()` is called. Currently, this data transfer is implemented using nonblocking sends and receives (e.g., using functions

`MPI_Isend()`, `MPI_Irecv()`, and `MPI_Waitall()`). So, in Phase 1, the entire contents of matrix `mR` is transferred into a local buffer by explicitly sending the local portion to all other nodes and explicitly receiving a portion of the matrix from the other nodes. And, in Phase 2, matrix `mP` is transferred into local memory before `mmultiply()` is called. In both phases, only the local portions of the other two matrices are accessed, and the access is done using loads and stores to local memory. Therefore, in the MPICH version, there is no need for data communications within the `mmultiply()` function itself.

Performance. The performance of the three different implementations of matrix multiplication is shown in Figs. 4 and 5. In both figures, the top graph shows the absolute speedups achieved by the systems for 2, 4, 8, and 16 processor nodes. The bottom graph shows the speedups normalized such that the MPICH speedup is always 1.00. For both the 512×512 and 704×704 datasets, all three systems achieve high absolute speedups for up to 8 processors. For up to 16 processors, both TreadMarks and Aurora show high speedups, but MPICH suffers in comparison. We discuss the MPICH results below. Overall, the high speedups are not surprising since matrix multiplication is known to be an easy problem to parallelize.

The performance difference between ideal speedup (i.e., unit linear) and the achieved speedup is generally due to the overheads of communicating matrix `mB` and, to a lesser extent, due to the need for barrier synchronization in the parallel programs. Speedups of between 13 and 14 on 16 processors are encouraging considering the relatively small datasets and the particular hardware platform. Even though the 704×704 dataset requires 250.7 seconds of sequential execution time, it is still a relatively small computational problem. For example, perfect unit linear

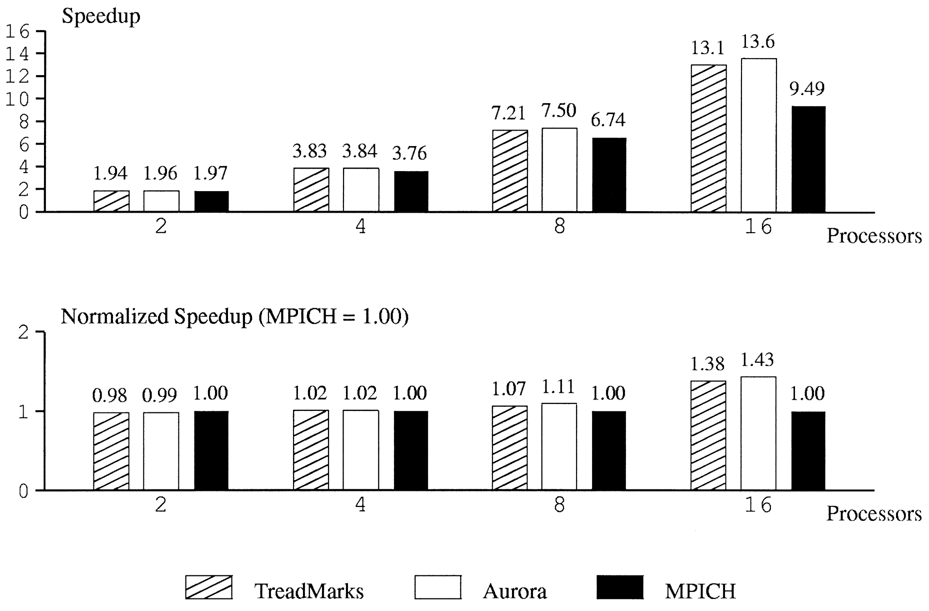

FIG. 4. Speedups for matrix multiplication, 512×512.

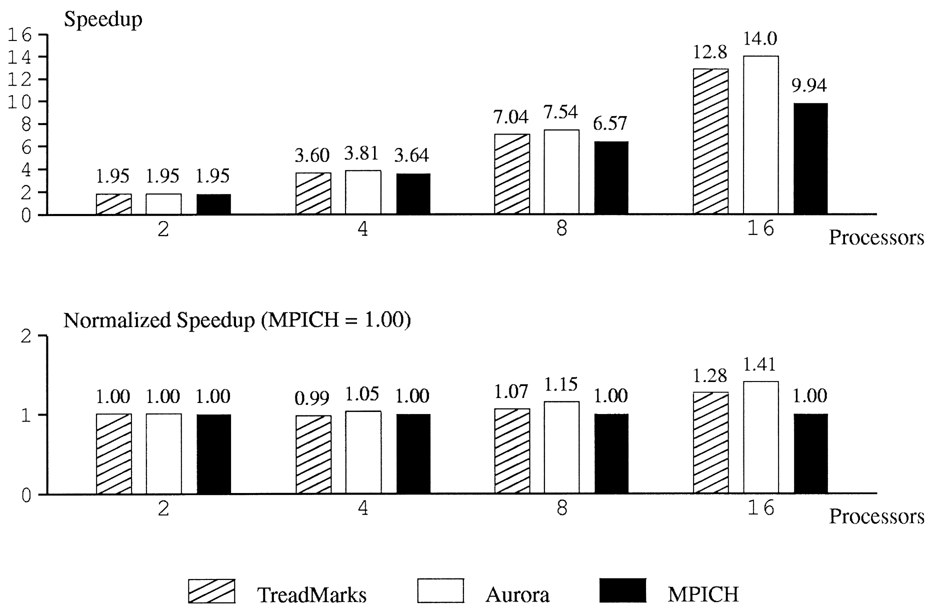

FIG. 5. Speedups for matrix multiplication, 704 × 704.

speedup on 16 processors requires a run time of 15.7 seconds for the 704 × 704 dataset. The fastest run time for that data point is achieved by the Aurora program at 17.9 seconds for a speedup of 14.0. The real time difference between ideal and achieved speedup is a fairly low 2.2 seconds, but that translates into a reduction of 2.0 in the absolute speedup.

Also, in recent years, processors have increased in speed at a greater rate than networks have increased in speed. In effect, the granularity of work has actually decreased (i.e., become worse for performance) for a given application and the absolute speedups of 15 (or better) on 16 processors reported in previous papers cannot be directly compared with these absolute speedups. As the number of processors increases (i.e., processor scalability), Aurora maintains a consistent, small, but growing, performance advantage over TreadMarks. This difference is attributable to how the two systems handle bulk-data transfer and data consistency.

As previously discussed, Aurora exploits the semantics of the scoped behaviour for a read cache to aggressively push data into remote caches. In contrast, TreadMarks transfers data using a request-response protocol that is invoked per-page and on demand. Although there is a potential for increased contention during the bulk-data transfer, as compared with a request-response protocol, our experimental results show that bulk-data transfer results in a net benefit for this data-sharing pattern. Also, since TreadMarks endeavours to provide a general-purpose data-consistency model, the protocol overheads of fault handling, twinning, diffing, and communication [13] are more expensive than the simpler approach taken in Aurora. There is no consistency model *per se* in Aurora; rather, data are transferred, manipulated, and made consistent on a language scope basis according to the create-use-destroy model of shared-data objects.

As the problem size increases (i.e., problem scalability), the performance gap between TreadMarks and Aurora also increases. Whereas Aurora's speedups are higher using 8 and 16 processors for the 704×704 dataset than for the 512×512 dataset, TreadMarks's speedups are consistently lower for the larger problem size. For example, the normalized speedup for TreadMarks falls from 1.38 to 1.28 on 16 processors between the two datasets. Normally, as the problem size increases, the granularity of work increases and speedups should also increase. This is especially true of matrix multiplication since the computational complexity grows $O(n^3)$ and the communication overheads grows $O(n^2)$. However, the lower speedups for the larger problem suggests that there may be a bottleneck within TreadMarks. One possible explanation is that the larger problem requires more shared pages, which may result in more contention for the network, fault handling, and protocol processing, as the larger matrix is demanded into each node. Also, the request-response communication pattern used by TreadMarks exposes the entire per-page network latency to the application, whereas the bulk-data transfer capability in Aurora creates a pipelined exchange of data to hide more of the latency. Part of Aurora's design philosophy is to try and avoid bottlenecks due to contention by keeping the data-sharing protocols as simple as possible (i.e., smaller handling and protocol overheads) and by supporting custom protocols, as with bulk-data transfer (i.e., smaller incremental cost as amount of data increases).

In fairness, it should be noted that newer research versions of TreadMarks include support for prefetching data, which may improve the performance of bulk-data transfer. However, these versions of TreadMarks are not available for use in this evaluation. And, even with prefetching, the consistency protocol overheads remain an inherent part of TreadMarks.

The performance of MPICH begins to lag behind the performance of TreadMarks and Aurora starting (marginally) at 4 processors, with the gap increasing at 8 and 16 processors. For 16 processors, the normalized speedups for Aurora are between 41% and 43% higher than for MPICH. First, we quantify and compare the data communication overheads in Aurora and MPICH. The same analysis is not performed for TreadMarks because it would require substantial modifications to the TreadMarks source code. Second, we consider some possible explanations as to why the overheads are higher for MPICH.

We isolated and measured the data-sharing overheads associated with matrix `mB` in function `mmultiply()` for both the Aurora and MPICH programs. The results for the 704×704 dataset are shown in Fig. 6 in terms of the number of seconds of real time of overhead. Lower times imply less data-sharing overhead. These real times are the average of five runs. Note that for the 16 processor case using Aurora, each processor sends (i.e., local data to all nodes) *and* receives (i.e., remote data from all nodes) approximately 1.77 MB of data, for a total of 3.54 MB of network input/output for each read cache of matrix `mB`. Since there are two phases and two read caches, a total of 7.09 MB of data is transferred per-node for each run. Of course, the amount of data transferred is the same for MPICH.

Recall that the Aurora program applies the read cache scoped behaviour to matrix `mB`. Two barriers were added to the Aurora program: one barrier just before

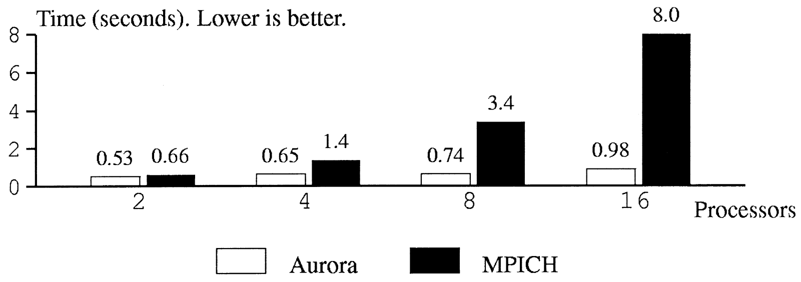

FIG. 6. Data-sharing overheads in matrix multiplication, 704 × 704, Aurora versus MPICH.

and one barrier just after the `NewBehaviour` macros for matrices `mA`, `mB`, and `mC` in `mmultiply()`. The execution time between the barriers is taken to be the data-sharing overhead since there is no more communications after the scoped behaviours have been applied. Although all the behaviours are measured, the overhead of the read cache dominates the reported times. Normally, these barriers are not required because a process can proceed with the multiplication as soon as its own local cache is ready, regardless of whether any other process is also ready to proceed. By adding the barriers to this experiment, we measure the worst case times for *all* processes to load their read cache.

Recall that the MPICH program loads the contents of matrix `mB` before calling `mmultiply()`. So, by measuring the amount of time required for this localized set of sends and receives, we obtain the total data-sharing overhead. As with the Aurora program, we added a barrier before and after this data exchange phase of the program, and the time between the barriers is taken to be the total data-sharing overhead.

Figure 6 shows that the real-time overheads of Aurora are between 12% (0.98 seconds versus 8.0 seconds) and 80% (0.53 seconds versus 0.66 seconds) that of MPICH for this particular data-sharing pattern. In the 16 processor case, the MPICH overheads are over eight times higher than the Aurora overheads. For MPICH, the overheads more than double as the number of processors doubles, which suggests a significant bottleneck as the degree of parallelism is scaled up. In contrast, Aurora's overheads grow at a rate that is less than the increase in the degree of parallelism.

Pinpointing the exact bottleneck in the MPICH program is difficult because a number of software and hardware layers are involved and not all of these layers (e.g., AIX's implementation of TCP/IP) are open for analysis. In the following discussion, we rely on our hands-on experience, the experimental evidence, and previously published research to posit some possible explanations. We theorize that Aurora's bulk-data transfer protocol for this type of sharing outperforms MPICH for two main reasons: First, Aurora's use of UDP/IP avoids some of the protocol overheads associated with TCP/IP. Second, Aurora avoids some of the overheads associated with a lack of data buffering in MPICH. Note that Aurora continues to use MPICH (and, thus, TCP/IP) for non-bulk-data transfers.

By using UDP, Aurora bypasses TCP's congestion avoidance algorithms and flow control mechanisms [3]. In an all-to-all data-sharing pattern, there *will be* congestion and contention somewhere in the system, regardless of the parallel programming system that is used. All processes are communicating at the same time and contend for resources such as the network, the network interface, and the network stack in the operating system. 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. This assumption is not generally valid on a wide area network (WAN), but it is valid in our cluster environment. If a network is not shared, waiting before retransmission wastes more network bandwidth through idleness than it saves in avoiding further congestion. It is also possible that TCP's flow control mechanisms are degrading performance for this sharing pattern. The interaction between TCP's various mechanisms, such as positive acknowledgments, windowed flow control, and slow start, can be complex, especially for our data-sharing pattern [17, 18]. Without access to the AIX internals, it is difficult to be more conclusive. However, in summary, TCP's robust and conservative approach to flow control is well-suited for shared WANs, but it is not necessarily optimal for dedicated LANs, such as for our applications.

We note that TreadMarks also uses UDP/IP for its network protocol and TreadMarks's performance is much closer to Aurora than to MPICH. Although TreadMarks's use of UDP is quite different from Aurora, the similarities in speedups between these two systems suggest that the main bottleneck is probably either MPICH or TCP and is not the physical network. If the bottleneck lies within TCP, then a rewrite of MPICH to use UDP for bulk-data transfer may close the performance gap between all three systems. However, a UDP-based version of MPICH is not currently available and we could not test this hypothesis. But, based on the experimental evidence, the performance problems with MPICH are likely the result of an unfortunate bottleneck instead of a fundamental design flaw with message passing or MPICH.

Why are the performance problems with TCP/IP not better known? Many of the published performance numbers for MPICH and TCP/IP are for single sender and single receiver sharing patterns on a dedicated network. The simple sender-receiver pattern and the dedicated network likely avoids any of the situations in which congestion avoidance and flow control mechanisms come into play.

As for MPICH's approach to data buffering, our experience is that large data transfers, especially in all-to-all patterns, must be fragmented and de-fragmented by the application programmer through multiple calls to the send and receive functions. As per the MPI standard, buffering is not guaranteed for the basic `MPI_Send()` and `MPI_Recv()` functions. Therefore, the MPICH programs suffer from the additional overheads of fragmentation for large data transfers. Admittedly, this problem could *potentially* be addressed in an alternate implementation of MPI. Or, different buffering strategies can be tried using other versions of the send and receive functions. In fact, a number of different strategies were tried without achieving better results.

3.4. Parallel Sorting by Regular Sampling

Design and Implementation. The Parallel Sorting by Regular Sampling (PSRS) algorithm is a general-purpose, comparison-based sort with key exchange [7, 15]. As with similar algorithms, PSRS is 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. 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.

Conceptually, a multi-phase algorithm with several shared-data objects, like PSRS, is particularly well-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 4 summarizes the phases of PSRS and the main Aurora optimizations. Note that the data movement in Phase 3 is implemented with `distmemcpy()`, which is a `memcpy()`-like construct that transparently handles shared-data vectors, regardless of the distribution and location of the local bodies. Function `distmemcpy()` is invoked by the reader of the data and is another example of asynchronous or one-side communication since it does not require the synchronous participation of the sender. The function interacts directly with the daemon thread on the sender's node.

TABLE 4

Per-Context and Per-Object Optimizations in Aurora's PSRS Program

Phase	Algorithm	Main optimizations in Aurora
1	Sort local data and gather samples.	Sort local data using owner-computes. Samples are co-located with master and are gathered using release consistency.
2	Sort samples, select pivots, and and partition local data.	Master sorts samples using owner-computes. Pivots are accessed from a read cache.
3	Gather partitions.	Partitions are gathered into local memory using `distmemcpy()`.
4	Merge partitions and update local data.	Merge partitions in local memory. *Note*. Partitions are merged, NOT sorted.

In the Aurora program, some optimizations are based on the programmer's knowledge of the application's data-sharing idioms. For example, there is a multiple-producer and single-consumer sharing pattern during the gathering of sample keys in Phases 1 and 2 (see [9]). This pattern has also been described as a gather operation [16]. Note that the vector `Sample` has been explicitly placed on processor node 0. In Phase 1, the samples are gathered by all processor nodes and optimized using release consistency. Each processor node updates a disjoint portion of the vector. In Phase 2, the master node (i.e., processor node 0) sorts all of the gathered samples. Since `Sample` is co-located 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 in the Aurora program. 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 processor 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 behaviour allows the programmer to optimize disjoint, update-only data-sharing idioms. Second, since the local body for `Sample` is explicitly placed on processor 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.

An explicit optimization is the bulk-data transfer protocol that is part of `distmemcpy()`. It is presumed that `distmemcpy()` is primarily used for transferring large amounts of data, so a specialized protocol is justified. In contrast to the "always exchange all the data" (i.e., "allgather") semantics and all-to-all sharing of a read cache in matrix multiplication, `distmemcpy()` uses a one-to-one protocol; there is only one receiver of the data transfer. As with the previous bulk-data protocol, the new protocol uses UDP instead of TCP and improves performance by adopting a more aggressive flow control strategy. And, as with the read cache optimization, bulk-data transfer has a net benefit despite the possibility of increased contention during the data transfer.

The TreadMarks program is a direct port of the original shared-memory program for PSRS [7]. All of the shared data structures reside on shared-memory pages and the basic demand-paging mechanisms of TreadMarks react to the changing data-access patterns of each phase. Unlike Aurora, there are no mechanisms for avoiding the protocol overheads for write-only data and to eagerly push data to the node that will use it in the next phase (i.e., `Sample` in Phases 1 and 2). However, for the 6 and 8 million datasets used in this evaluation, the main determinant of performance is the efficiency of the data transfer in Phase 3, as we will see below.

The MPICH program implements with explicit sends and receives what Aurora implements with the scoped behaviour and data placement strategies described above. For example, like Aurora and unlike TreadMarks, the MPICH program does not perform a message receive for the contents of `Sample` in Phase 1 before updating them. The gathered sample keys are sent eagerly with a single message send in preparation for Phase 2.

Performance. The performance of PSRS is given in Figs. 7 and 8. The MPICH version of PSRS is faster than both Aurora and TreadMarks for the 2 processor data points. The TCP-based data transfer in MPICH is very effective when there is relatively low contention, such as when only 2 processors exchange data. However, as the number of senders and receivers increases with the degree of parallelism, both TreadMarks and Aurora begin to significantly outperform MPICH. Benefiting from the UDP-based bulk-data transfer protocol, Aurora achieves the highest performance for all the 4, 8, and 16 processor cases.

Except for the 2 processor data points, Aurora is consistently faster than TreadMarks by margins of up to 25%. As with matrix multiplication, the performance difference grows with the size of the problem because Aurora's bulk-data transfer protocol is better at pipelining larger data transfers than TreadMarks's request-response protocol.

The performance advantage of Aurora over MPICH grows with both the degree of parallelism and the size of the dataset. With more processors, there is more contention for various hardware and software resources. With more keys to sort, Phase 3 grows with respect to the amount of data transferred. For the smaller 6 million key dataset, Aurora is 52% faster than MPICH when using 16 processors.

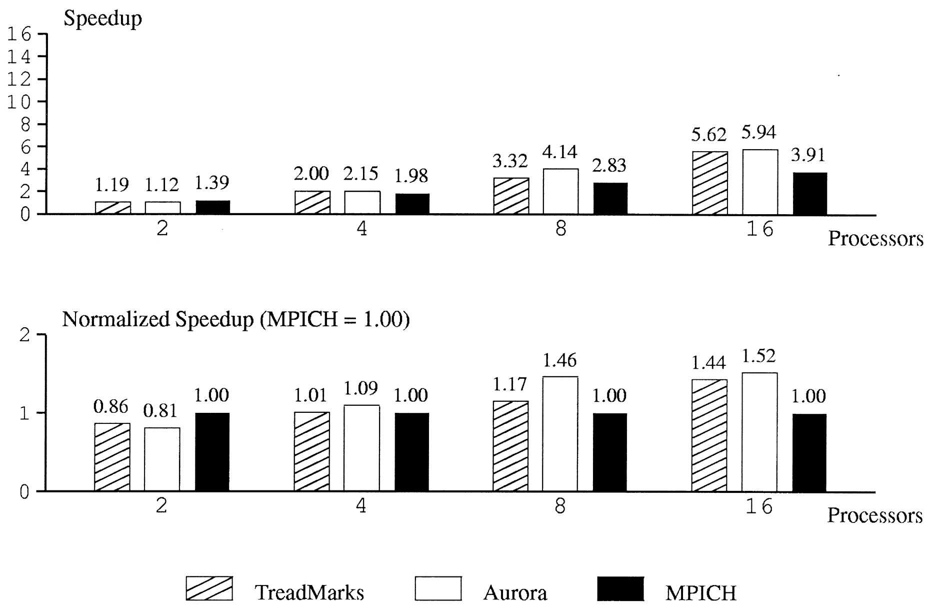

FIG. 7. Speedups for parallel sorting (PSRS), 6 million keys.

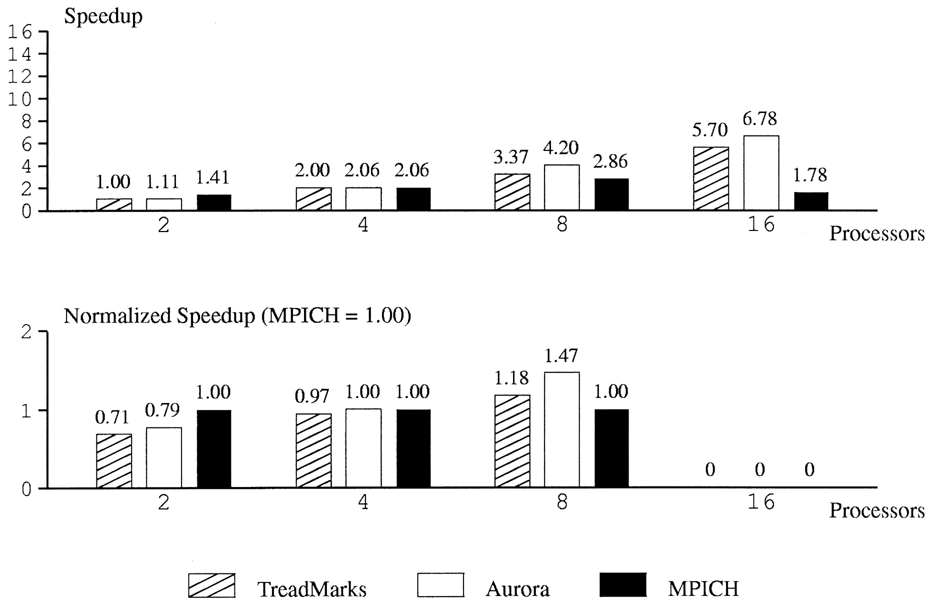

FIG. 8. Speedups for parallel sorting (PSRS), 8 million keys.

For the larger 8 million key parallel sort, Aurora is 380% faster than MPICH on 16 processors. The normalized speedups bars for 16 processors and the 8 million key dataset are not shown in Fig. 8 because they are too large for this pathological data point.

The problem with MPICH (and TCP) for this application is even worse than these numbers indicate. The variations in the real times of the MPICH program on 16 processors are so large that the reported numbers are the *minimum* values over five runs, instead of averages. The average values are up to 40% higher than the minimum values. The problem, once again, is large data transfers between multiple senders and receivers. As we saw previously with matrix multiplication, Aurora and TreadMarks are UDP-based and avoid the performance problems with the TCP-based MPICH for these data transfers. Interestingly, the original implementation of `distmemcpy()` in Aurora did not have a UDP-based bulk-data transfer protocol and it suffered from the same high variability and low performance problems exhibited by MPICH. When the bottleneck was identified, the new bulk-data transfer protocol was implemented in the Aurora software layers and without any changes to the PSRS source code.

4. 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. 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. What distinguishes Aurora is its use of scoped behaviour to provide per-context and per-object flexibility in applying data-sharing optimizations.

Scoped behaviour is both an API to a set of system-provided data-sharing optimizations and an implementation framework for the optimizations. As a framework, one advantage of scoped behaviour is how it carries semantic information about specific data-sharing patterns across software layers and enables specialized per-object and per-context protocols. We have described how, when loading a read cache in matrix multiplication, the scoped behaviour is specified by the application programmer. Furthermore, the knowledge that all processes must participate in the bulk-data transfer and all of the data must be transferred is passed down to Aurora's run-time layer and exploited using a bulk-data transfer protocol. A similar protocol is used when exchanging keys in a parallel sort.

In a comparison of the two applications implemented using Aurora, TreadMarks, and MPICH, the performance of Aurora is comparable to or better than the other systems. MPICH appears to suffer performance problems due to its reliance on TCP/IP for bulk-data transfers, even in a high-contention all-to-all pattern. TreadMarks suffers from its reliance on a request-response data movement protocol. In contrast to MPICH, Aurora uses UDP/IP for bulk-data transfers to avoid the protocol bottlenecks of TCP/IP. In contrast to TreadMarks, the scoped behaviour pipelines the data transfer to avoid the message and latency overheads of TreadMarks's request-response protocol. The pipelining is possible because the scoped behaviour encapsulates the fact that all of the data must be transferred and the system can be more proactive instead of reactive. Consequently, on a network of workstations connected by an ATM network, Aurora generally outperforms the other systems in the situations where a bulk-data transfer protocol is beneficial.

ACKNOWLEDGMENTS

Thank you to the anonymous referees for their comments. This work was part of my Ph.D. research 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. This work has been supported by an NSERC Operating Grant and a research grant from the University of Alberta.

REFERENCES

1. 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 Comput*. **29**(2) (February 1996), 18–28.
2. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, Orca: A language for parallel programming of distributed systems, *IEEE Trans. Software Engrg*. **18**(3) (March 1992).
3. H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz, TCP behavior of a busy internet server: Analysis and improvements, *in* "Proceedings of IEEE Infocom, San Francisco, CA, March 1998."

4. J. K. Bennett, J. B. Carter, and W. Zwaenepoel, Munin: Distributed shared memory based on type-specific memory coherence, *in* "Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming," Assoc. Comput. Mach., New York, 1990.
5. J. O. Coplien, "Advanced C++: Programming Styles and Idioms," Addison–Wesley, Reading, MA, 1992.
6. N. E. Doss, W. D. Gropp, E. Lusk, and A. Skjellum, "A Model Implementation of MPI," Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1993.
7. X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi, On the versatility of parallel sorting by regular sampling, *Parallel Comput.* **19**(10) (October 1993), 1079–1103.
8. P. Lu, Implementing optimized distributed data sharing using scoped behaviour and a class library, *in* "Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS), Portland, Oregon, June 1997," pp. 145–158.
9. P. Lu, Using scoped behavior to optimize data sharing idioms, *in* "High Performance Cluster Computing: Programming and Applications" (R. Buyya, Ed.), Vol. 2, pp. 113–130, Prentice Hall PTR, Upper Saddle River, NJ, 1999.
10. P. Lu, Implementing scoped behaviour for flexible distributed data sharing, *IEEE Concurrency* **8**(3) (July–September 2000), 63–73.
11. P. Lu, "Scoped Behaviour for Optimized Distributed Data Sharing," Ph.D. thesis, University of Toronto, Toronto, ON, Canada, January 2000.
12. W. G. O'Farrell, F. Ch. Eigler, S. D. Pullara, and G. V. Wilson, ABC++, *in* "Parallel Programming Using C++" (G. V. Wilson and P. Lu, Eds.), MIT Press, Cambridge, MA, 1996.
13. E. W. Parsons, M. Brorsson, and K. C. Sevcik, Predicting the performance of distributed virtual shared-memory applications, *IBM Systems J.* **36**(4) (1997), 527–549.
14. H. S. Sandhu, B. Gamsa, and S. Zhou, The shared regions approach to software cache coherence, *in* "Proceedings of the Symposium on Principles and Practices of Parallel Programming, May 1993," pp. 229–238.
15. H. Shi and J. Schaeffer, Parallel sorting by regular sampling, *J. Parallel Distrib. Comput.* **14**(4) (1992), 361–372.
16. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, "MPI: The Complete Reference," MIT Press, Cambridge, MA, 1996.
17. W. R. Stevens, "TCP/IP Illustrated. Vol. 1. The Protocols," Addison–Wesley, Reading, MA, 1995.
18. G. R. Wright and W. R. Stevens, "TCP/IP Illustrated. Vol. 2. The Implementation," Addison–Wesley, Reading, MA, 1995.

PAUL LU is an Assistant Professor of Computing Science at the University of Alberta. His B.Sc. and M.Sc. degrees in Computing Science are from the University of Alberta. He worked on parallel search algorithms as part of the Chinook project. His Ph.D. (2000) is from the University of Toronto and is in the area of parallel programming systems. He has collaborated with researchers at IBM's Center for Advanced Studies (CAS) in Toronto. In 1996, he co-edited the book "Parallel Programming Using C++" (MIT Press). His current research is in the areas of high-performance computing and systems software for cluster computing. `http://www.cs.ualberta.ca/~paullu/`