Scoped Behaviour for Optimized Distributed Data Sharing

by

Chien-Ping Paul Lu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Scoped Behaviour for
Optimized Distributed Data Sharing

Chien-Ping Paul Lu
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2000

We introduce the novel *scoped behaviour* abstraction and examine how it is used to optimize distributed data-sharing patterns within the Aurora parallel programming system. Scoped behaviour is an application programmer's interface to a set of system-provided optimizations; it is also an implementation framework for the optimizations. Aurora is a distributed shared data system where shared-data objects are implemented in C++ as abstract data types. Aurora has been prototyped on a network of workstations connected by an ATM network.

The design, implementation, and evaluation of Aurora and scoped behaviour contributes to the field of parallel and distributed systems by demonstrating that:

1. Scoped behaviour can provide per-object and per-context (i.e., specific portion of the source code) flexibility when applying data-sharing optimizations. In contrast to some other systems, Aurora programs can be incrementally tuned and only a minimum number of error-prone changes to the source code are required in order to experiment with different optimization strategies.

2. Scoped behaviour can be implemented without language extensions and without special compiler support. Scoped behaviour's novel implementation framework can exploit both compile-time and run-time information about the parallel program.

3. A parallel programming system based on a high-level shared-data abstraction can achieve high performance. In a performance evaluation of four applications implemented using

three different types of parallel programming systems, Aurora usually matches or outperforms, sometimes by a wide margin, TreadMarks (a distributed shared memory system) and a message-passing system (either MPICH or PVM, depending on the application).

Acknowledgements

Many thanks to my supervisor, Songnian Zhou, and the other members of my thesis committee: Ken Sevcik, Michael Stumm, Tarek Abdelrahman, and Michiel van de Panne. In particular, Ken's leadership and support of the Parallelism on Workstations (POW) Project made it possible to complete the experimental evaluation.

Thanks also to Laxmikant Kalé for acting as my external examiner and for a record-breaking turnaround on his report.

No matter where I have called home, I have always been blessed with wonderful friends. My time in Toronto has been made a lot more fun and worthwhile thanks to fellow graduate students Jonathan Appavoo, Ben Gamsa, Eric Parsons, Karen Reid, and Daniel Wilk. Kathy Yen, our department's Graduate Admissions Officer, was a constant source of smiles.

Greg Wilson provided a great deal of valuable feedback on my research and this dissertation. Greg has been a friend in many ways.

My thanks to Jonathan Schaeffer for his continued support and his comments on this dissertation. Thanks also to Duane Szafron for his encouragement during the last stretch.

The financial support of NSERC and the Department of Computer Science is gratefully acknowledged.

Thanks to Bill O'Farrell, IBM, and the Center for Advanced Studies, for their support of the POW Project and this research. My involvement with and access to the ABC++ system was very helpful. The POW Project was also supported by ITRC.

Thanks to the Advanced Computing Laboratory at the Los Alamos National Laboratory, especially John Reynders, Kate Keahey, and Alan Keahey, for an enjoyable and productive time there in 1996. Good people, good food, and good code.

Lastly, thank you to my friends from residence, my friends in Edmonton, and my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel and distributed computing is a diverse area of research and practice [Wilson, 1995]. Distributed-memory platforms, such as a network of workstations, are attractive because of their ubiquitousness and good price-performance. However, achieving high absolute performance can be challenging. Sharing data across distributed memories is expensive, especially when compared to hardware-based shared memory, so it is important to have optimizations that reduce the number of network messages and other overheads.

Parallel applications can be developed using parallel languages (e.g., High Performance Fortran), libraries (e.g., PLAPACK), run-time systems (e.g., Message-Passing Interface), or a combination of these techniques. Each type of parallel programming 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.

Unfortunately, performance and usability are sometimes conflicting design goals for parallel programming systems. On the one hand, low-level control of communication operations can give explicit message passing a flexibility and performance advantage. On the other hand, systems based on shared-memory and shared-data models are becoming increasingly popular and widespread for parallel applications. Accessing local and remote data using the same programming interface (e.g., reads and writes) is often more convenient and less error-prone than mixing local accesses with explicit message passing.

Another influence on the research area is the fact that distributed systems and applications often have 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. For example, all message-passing systems provide general-purpose mechanisms to send and receive data across distributed processor nodes. However, some systems (e.g., Message-Passing Interface) also support commonly occurring group operations, such as all-to-all communication [Gropp *et al.*, 1994, Snir *et al.*, 1996]. By pre-packaging a data-sharing idiom into a single function, the system facilitates code reuse and it can also exploit the semantics of the idiom for higher performance.

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. More generally, it is often desirable to alter or optimize a program according to the needs of a specific portion of the source code (i.e., context), such as a particular loop or phase. Many parallel programs consist of multiple phases where the shared data structures being accessed and the computation being performed can be dramatically different from phase to phase. Therefore, an optimization that is appropriate for one use-context may be inappropriate for a different context. The flexibility to localize the optimizations is also useful when incrementally optimizing the program and when experimenting with different optimization strategies.

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) [Li, 1988, Bennett *et al.*, 1990b, Amza *et al.*, 1996] and *distributed shared data* (DSD) [Bal *et al.*, 1992, Sandhu *et al.*, 1993, Johnson *et al.*, 1995] 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 point to and 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. 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++ [Lu, 1997a, Lu, 1997b, Lu, 1999, Lu, 2000]. 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 datatypes. But, since each object is an independent unit of sharing and management, the data-access behaviour can be optimized on a per-object basis.

Aurora is unique in its use of the *scoped behaviour* abstraction for applying optimizations.[1] Scoped behaviour is an application programmer's interface to a set of system-provided optimizations; it is also a flexible and powerful implementation framework for a variety of data-sharing optimizations. With scoped behaviour, a new language scope (e.g., nested braces in C++) around selected source code can be used to optimize the data-sharing behaviour of specific objects within that scope. Different language scopes can be optimized in different ways, thus the optimizations can be applied on a per-object *and* per-context basis. 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 combined with data placement mechanisms, scoped behaviour makes it easy to implement and experiment with many different optimization strategies.

We begin with a discussion of background material and related work in parallel programming systems. We continue by introducing the various process and data models supported in the Aurora system. Through examples, we demonstrate how the application programmer can write complete parallel programs using Aurora. Then, we demonstrate how, once the programs are working correctly, the programmer can incrementally tune and optimize performance using scoped behaviour in a manner not possible in many other systems.

Another contribution of this research is the demonstration of how scoped behaviour can be implemented without language extensions and without special compiler support. We show how language scopes can be combined with object-oriented design principles to create a novel implementation framework for scoped behaviour.

We have designed, implemented, and evaluated the performance of Aurora and scoped behaviour on a network of workstations connected by an ATM network. Four applications have been implemented using Aurora, TreadMarks (i.e., a DSM system), and either MPICH or PVM (i.e., a message-passing system). The applications are matrix multiplication, a 2-D diffusion simulation, the Parallel Sorting by Regular Sampling algorithm, and the Travelling Salesperson Problem. In all except for a few data points, the performance of Aurora is comparable to or faster than the performance of the other systems. In some situations, Aurora is several-fold faster. An analysis of Aurora's performance advantage shows that scoped behaviour, and the high-level semantics associated with it, allows for optimizations that are not available (or not possible) in other systems.

[1] We interchangeably refer to the scoped behaviour abstraction, the scoped behaviour mechanism, and the scoped behaviour technique. As we will see, scoped behaviour is a broad and high-level concept that has many different aspects to it depending on what layer of the software system is being discussed.

Chapter 2

Background and Related Work

Scoped behaviour and the other ideas embodied in Aurora draw upon a large base of research in parallel and distributed systems. Much of the terminology and many of the concepts found in parallel programming environments have their origins in parallel hardware. Therefore, to properly place this work in context, we review basic and advanced ideas in parallel hardware and software. We also motivate the design of Aurora and the role of scoped behaviour by considering the notion of recurring structures and patterns, called idioms, in parallel applications.

2.1 Basic Hardware Concepts

Parallel hardware can vary from distributed-memory multicomputers, in which the memory of other processors cannot be directly accessed, to shared-memory multiprocessors, in which load and store instructions can access any memory location. A *network of workstations* (NOW) [Anderson *et al.*, 1995] is an example of a distributed-memory multicomputer while a bus-based *symmetric multiprocessor* (SMP) is an example of a shared-memory multiprocessor.

Typically, a NOW consists of *processor nodes* (or simply *nodes*) that combine both a processor and cache (sometimes referred to as a processor element or PE) with local physical memory (Figure 2.1). The processors can access their local memories through load and store operations, but remote memories must be accessed using network messages. Of course, when the hardware does not support a load-store shared-memory environment, the software can simulate a shared-data model, but with some performance penalty due to software overheads.

There are a number of possible architectures for hardware-based shared-memory multiprocessors, usually varying on where the shared memory is physically located in the machine. Depending on whether the memory is centralized (Figure 2.2) or distributed, the speed of accessing different portions of memory can vary. In all cases, the key characteristic of hardware-based shared memory is that the interconnection network allows loads and stores to the shared

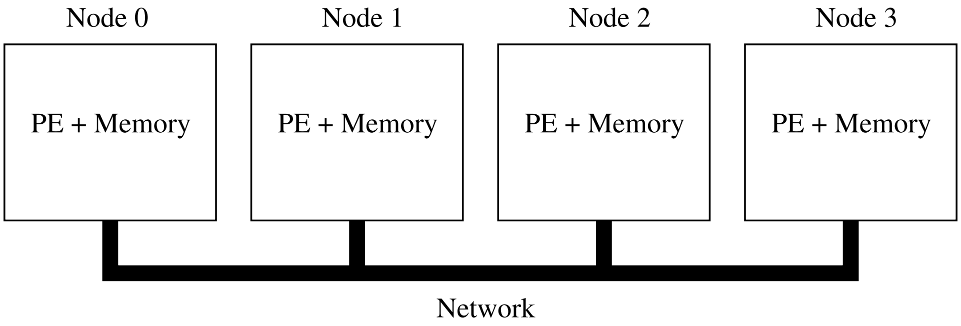


Figure 2.1: Network of Workstations

memory. In contrast to distributed-memory architectures, network messages are not required to access remote data.

If the latency of accessing every physical memory location from any processor is constant, then the machine is said to have a *uniform memory access* (UMA) architecture. However, some computer architectures use scalable interconnection networks, such as meshes [Lenoski *et al.*, 1992, Agarwal *et al.*, 1995] and fat trees [Thi, 1992], in which independent communication links increase bandwidth, but the distance between individual processors can vary. Computer architects also build large-scale multiprocessors using a hierarchy of networks. For example, one could have meshes connecting buses [Lenoski *et al.*, 1992, Kuskin *et al.*, 1994] or rings connecting buses [Vranesic *et al.*, 1991, Abdelrahman *et al.*, 1994]. Having to traverse a network implies that the cost of accessing local memory is different from the cost of accessing remote memory, resulting in a *non-uniform memory access* (NUMA) architecture. Consequently, the issue of locality, whether the data being accessed by a process is in local or remote memory, is important to achieving high performance on NUMA multiprocessors.

To improve memory access performance, modern computers usually include fast cache memory at one or more levels of the storage hierarchy. Caches enhance performance by both reducing latency and increasing bandwidth to data whenever the load-store operations hit a data item currently in the cache. Although caches differ in capacity and internal organization, their main goal is to improve performance when there is locality in memory accesses. However, if different processors are allowed to cache the same data, there may be multiple copies of the data in the system. If the data is read-write, then the different caches need to be kept consistent. The *cache coherency problem* is one of how to consistently propagate the updated shared value, or how to consistently invalidate old copies of the data, across the various caches and main memory. A *cache coherency protocol* implements the consistent propagation or invalidation of shared data using the specific mechanisms of the processor, the capabilities of

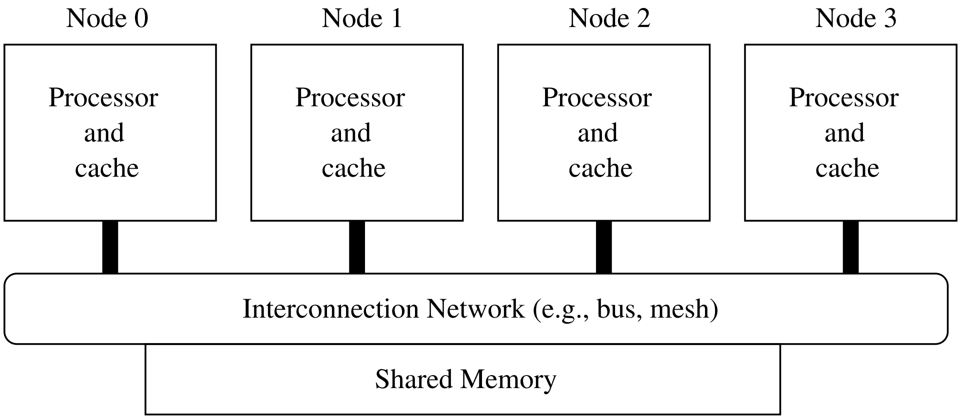


Figure 2.2: Hardware-Based Shared Memory

---

the interconnection network interface, and the topology of the network.

How and when a value is considered to be globally visible depends on the logical *memory consistency model*. Independently of whether caches are present, computer systems must support a particular consistency model. Four different models of memory consistency are *sequential consistency* [Lamport, 1979], *processor consistency* [Goodman, 1989], *weak consistency* [Dubois *et al.*, 1986], and *release consistency* [Gharachorloo *et al.*, 1990]. The models mainly differ in the allowable ordering of load and store operations to memory. Since maintaining a strict ordering of load and store memory accesses in a concurrent and distributed system can be expensive, the purpose of relaxing the ordering is to reduce the ordering overheads and improve performance. Sequential consistency requires a strict global ordering of memory operations, such that loads and stores complete in the same order they are issued, and all processors observe the same order of operations to shared data. Typically, this implies that many loads and stores will block the processor so that the strict order of their completion can be enforced. Processor consistency allows a load operation to bypass a store operation to an independent data item, thus potentially requiring fewer stalls and resulting in higher performance.

Relaxed consistency models, such as weak consistency and release consistency, interleave synchronization operations with the loads and stores so that the ordering requirements can be relaxed *between* synchronization points. For example, release consistency distinguishes between *acquire synchronizations* (comparable to lock operations) and *release synchronizations* (comparable to unlock operations). All the loads and stores issued after an acquire synchronization must block until the acquire completes, while a release synchronization must block until all previous loads and stores complete. Furthermore, acquire and release synchronizations must be processor consistent, as in processor consistency, between themselves. Therefore, although the rules for ordering operations between the acquire and release synchronizations are relaxed, the synchronizations themselves must be more strictly ordered.

There is a direct mapping between the traditional notion of using locks and barriers in concurrent programs and the release consistency model. By associating an acquire synchronization with a lock operation, a release synchronization with an unlock operation, and a release-acquire pair with a barrier synchronization, the hardware can still guarantee sequential consistency semantics for properly synchronized programs (i.e., no race conditions). Except at the synchronization points, release consistency allows the hardware to better pipeline load and store operations. In this way, release consistency improves performance by helping to hide the latency of remote operations. However, the synchronization points themselves are expensive and should be avoided if possible in order to maximize performance.

In general, if a specific portion of the program or context is special in some way, it makes sense to identify the context to the systems software and hardware. By labelling and associating some high-level semantics with the context, the system may be able to optimize performance. For example, code that accesses shared data is bracketed by a lock-unlock (and acquire-release) pair to create a critical section. On modern computers, the lock (and unlock) mechanism requires special support and action on the part of the systems software and hardware. By the semantics of lock-unlock and mutual exclusion, no other process should be accessing the shared data at the same time as the process with the lock, therefore the order in which other processes view the loads and stores does not matter until the next synchronization point. Thus, a relaxed ordering between synchronizations can increase performance, but it does not change the correctness of a program.

Similarly, scoped behaviour uses language scopes to bracket or select a context. Within the language scope, scoped behaviour attempts to optimize performance without changing the correctness of the code. But, whereas acquire-release synchronizations and locks are dynamic, language scopes are static. Locks and other synchronization mechanisms are usually functions calls to a system library and are treated by the compiler in the same way as any other function call. With dynamic mechanisms, it is not always possible to determine which lock is matched with which unlock just by inspecting the source code; the compiler cannot easily match one function call with another function call. Especially for complex locking strategies and source code with many control flow changes, it can be difficult to reason about the behaviour of the program and there are many errors that cannot be detected until run-time. With static mechanisms, it is clear from the source code which context is affected by the scoped behaviour. In fact, if a language scope is not properly nested and bracketed, the compiler will detect the error at compile-time. And, as we will see, scoped behaviour can be extended to also exploit run-time information for more optimizations. Easily selecting source code contexts, integrating a context's semantics across software and hardware layers, and exploiting those semantics for higher performance are all important aspects of the research into scoped behaviour.

2.2 Parallel Programming Systems

We review and critique a number of parallel programming systems as to their performance-oriented features and safety for a variety of parallel idioms. *Safety* is the ability to prevent or avoid programming errors in a programming system. For example, forgetting to match a lock with an unlock is a common programming error that is not detected by many systems, so these systems are lacking in that aspect of safety. *Idioms* are a general and high-level concept that refers to recurring patterns or structures that can be identified in parallel programs. For example, process pipelines and all-to-all data exchanges are common idioms in many different parallel programs. Most contemporary systems provide fast mechanisms and efficient policies, but many are limited in their flexibility, safety, and support for common parallel idioms.

We have identified three common weaknesses of current systems that our research addresses:

1. *The abstractions are too low level.* Many programming systems are designed around specific mechanisms (e.g., message sends and receives), instead of around problem-solving techniques. The programmer is largely responsible for correctness and performance tuning.

   The need for higher levels of abstraction is well known, but low-level programming systems continue to dominate the literature. One important exception is the work on data-parallel languages, such as High Performance Fortran (HPF) [Koelbel *et al.*, 1994].

2. *The interfaces are too low level.* Sometimes, the abstractions are fairly high level (e.g., shared data on multicomputers), but the interface to the abstraction is too low level (e.g., load-store operations). With a low-level interface, it is more difficult for the system to transparently change the abstraction's implementation to use more efficient mechanisms and policies.

   Some researchers have argued for high-level interfaces [Levelt *et al.*, 1992], but the need does not seem to be generally accepted. In fact, some systems resort to complex and expensive strategies to support high-level abstractions with low-level interfaces. The desire to support existing shared-memory applications influences many contemporary systems, including the software-based distributed shared memory systems discussed below.

3. *There is poor integration of abstractions and interfaces through all the layers of software and hardware.* Safety when using the abstractions requires a variety of static and dynamic information. Relying only on either compile-time or run-time information often results in gaps in safety.

   For example, we have already seen how locks are a dynamic and run-time-only mechanism. The proper use of lock-based synchronization is entirely up to the programmer and the compiler cannot check the correctness of the critical section. The compiler treats the function calls to the synchronization library in the same way as other function calls, therefore it cannot check or enforce that every lock is matched with a corresponding unlock. If the compiler can be made aware of the significance of the synchronization through better integration, it can generate code to always ensure that a lock is freed at the end of a function, which enhances the safety of locks.

   The need for safety is widely accepted, but many systems do not address the problem. One argument against tighter integration of the layers is that it reduces the portability of the system. However, in parallel computing, concurrency and non-determinism make application development significantly more difficult and error-prone, so the issue of safety must be addressed.

2.2.1 Message Passing: MPI

Message-passing systems, such as the Message-Passing Interface (MPI) [Gropp *et al.*, 1994, Snir *et al.*, 1996], provide the three basic mechanisms of parallel programming: processes, data channels, and operations to send and receive messages on the channels. Notably, each message send must be matched with a corresponding message receive (Figure 2.3). Programs written using MPI consist of processes in separate address spaces that communicate using messages of contiguous byte data. No specific structure is imposed on the messages, so the programmer is free to define messages of arbitrary content and arbitrary length. By explicitly exchanging messages, two processes can share data and synchronize their execution. However, the explicit nature of message passing suffers from the same problems as explicit locks and unlocks: it can be hard and error-prone to match a send with the proper receive in programs with complicated control flow changes.

Typical message-passing systems consist of both a function library and a run-time system. The library interfaces with the run-time system to create and manage process, channel, and message resources. After initialization, the processes use system-provided send and receive functions to pass messages. The libraries provided with message-passing systems can be extensive. For example, the MPI library provides functions for broadcasting messages and group

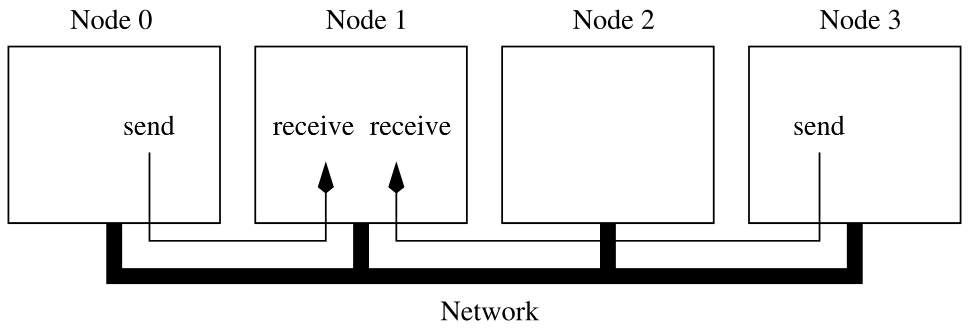


Figure 2.3: Message-Passing Architecture

communication. In total, there are 125 different functions in the comprehensive MPI library. Other message-passing systems, most notably the Parallel Virtual Machine (PVM) [Geist *et al.*, 1994], also exist, but the core functionality of sending and receiving messages remains the same.

One advantage of message-passing systems is that they can be ported to almost any parallel hardware platform. Only basic communication mechanisms are required of the underlying hardware, therefore parallel programs written using MPI can be more easily ported to a diverse range of parallel computers, from workstations, to hypercubes, to the Cray T3D. The T3D has hardware support for shared memory, but MPI is used to support existing applications.

One disadvantage of message-passing systems is that the functionality is provided at a low level of abstraction. The functionality is sufficient to develop any parallel application, but the coding effort required can be substantial. Typically, the programmer is responsible for even the most mundane and error-prone tasks, such as marshaling and de-marshaling messages of different datatypes. Of course, library functions in MPI raise the level of abstraction of some operations, but they cannot properly address the safety issue because libraries do not have access to the necessary compile-time information. Furthermore, for MPI applications to be truly performance-portable, they must avoid certain operations and library functions that cannot be efficiently implemented on all platforms. A performance-oriented MPI programmer must consider message buffering and implicit synchronization issues [Saphir, 1994]. Therefore, in practice, it is possible to achieve high performance with MPI, and other message-passing systems, but the programmer must be aware of potential safety problems and implementation-specific performance issues.

2.2.2 Hardware Shared Memory and Cache Coherence: COOL

In sequential programming, parameter passing by reference and the use of global variables depend on the ability to share data across the function call interface, so most programmers are already well-versed in the concept of sharing memory. Therefore, shared-memory systems are more familiar and convenient than explicit message-passing systems. However, supporting cache-coherent shared memory in hardware is difficult because of scalability problems. Applications that frequently access remote memory or have poor cache reuse will suffer from poor performance. Therefore, given a shared-memory multiprocessor, a programming system should also support locality optimizations.

Often, the available performance optimizations are determined by the underlying hardware. For example, the DASH multiprocessor [Lenoski *et al.*, 1992] has influenced the support for locality optimizations in the Concurrent Object Oriented Language (COOL) [Chandra *et al.*, 1994]. Other programming environments exist for DASH (e.g., the ANL parallel macros [Boyle *et al.*, 1987] and Jade [Rinard *et al.*, 1993]) and other cache-coherent shared-memory multiprocessors have also been built (e.g., Alewife [Agarwal *et al.*, 1995] and NUMAchine [Abdelrahman *et al.*, 1994]), but we focus on DASH and COOL as an interesting and representative case study.

DASH is a cache-coherent shared-memory parallel computer consisting of interconnected clusters of multiprocessors. Two mesh networks are connected to a custom interface board that occupies a slot on the shared bus in each cluster. The hierarchy of mesh-connected buses results in a NUMA architecture. The logical address space presented to the programmer is shared and coherent at a cache line granularity between all related processes. Within a cluster, memory coherency is maintained by the hardware using a snoopy bus-based coherence protocol [Gharachorloo *et al.*, 1990]. Between clusters, the meshes and custom interface boards keep shared data consistent using a hardware implementation of a full-map directory-based protocol [Lenoski *et al.*, 1990].

Through language extensions to C++ and a run-time system, COOL provides the mechanisms typically required of shared-memory programming systems, as well as locality management mechanisms. Parallel functions support the creation of parallel processes, mutex functions support mutual exclusion, condition variables support event synchronization and a `waitfor` construct supports barrier synchronization. COOL also allows the programmer to specify how data objects, processes, and processors should be co-located through various forms of the `affinity()` mechanism. For example, a parallel function can specify that it should be executed on the processor whose local memory contains an object pointed to by `this` with the specification `affinity(this, OBJECT)`. Also, a parallel function can specify that it should be scheduled back-to-back with other functions that operate on the same object, to improve cache reuse, with `affinity(this, TASK)`. The run-time system treats all affinity specifications as performance hints and tries to co-locate and schedule accordingly. Objects can also be migrated to specific processors with COOL's `migrate()` mechanism. Using the various locality optimization mechanisms, the performance of applications written using COOL can be substantially improved. The issue of data placement and co-location is also important in the Aurora parallel programming system. Although the ideas prototyped in COOL are noteworthy, the use of C++ language extensions in COOL has limited its impact to the research community.

The advantage of DASH's hardware support for both shared memory and cache coherence is performance. The hardware-based finite state machines and the optimized low-level protocols result in high performance. Also, DASH uses release consistency to help hide the latencies of remote memory operations. By allocating locks from a special portion of DASH's memory, the normal synchronizations of a COOL application are visible to the hardware, so it can transparently associate the proper acquire and release synchronizations with the normal locking protocol [Gharachorloo *et al.*, 1990]. However, from a system designer's point of view, the support for cache-coherent shared memory has a cost in terms of design complexity and extra hardware for the multiprocessor.

A disadvantage of hardware-based multiprocessors, like DASH, is that only a single coherence protocol is typically supported. Certain shared-data access patterns are most efficiently implemented using different protocols [Bennett *et al.*, 1990a, Bennett *et al.*, 1990b], but only one protocol is supported in hardware. It is easier for software systems, such as Aurora, to support multiple protocols. One way to increase the flexibility of the hardware is to use a protocol processor instead of a finite state machine, as in the FLASH project [Kuskin *et al.*, 1994], so that different protocols can be downloaded and supported. Another solution is to turn off hardware-based cache coherence for selected regions of memory and allow software to manage coherence in those regions, as in the NUMAchine project [Abdelrahman *et al.*, 1994].

2.2.3 Software Cache Coherence: Shared Regions

Some multiprocessors, such as Hector [Vranesic *et al.*, 1991] and the BBN TC2000 [BBN, 1989], support shared memory in hardware, but do not support cache coherence in hardware. That is to say, loads and stores can be made to remote memory, but the hardware is not responsible for keeping the processor caches in a consistent state. As a result, early experiments with these machines included disabling the caches for shared data. However, the DASH experience shows that not caching shared data is a serious performance problem [Gupta *et al.*, 1991], so a software-based strategy is required.

The Shared Regions (SR) model is a software-only technique to support a relaxed consistency model of shared memory through a combination of an access control programming model and various software cache coherence protocols [Sandhu *et al.*, 1993, Sandhu, 1995]. The concept of access controls, which is a form of labelling of contexts, is related to both locks and scoped behaviour. The key performance benefit of the SR approach is to always allow the caching of shared data. Programs written for the relaxed memory model of SR can cache all shared data and the SR run-time system maintains data coherence in software. Although the current SR implementations are for the C and C++ programming languages, the SR model is not language-specific.

As with many other shared-data models, SR requires the programmer to insert synchronization primitives into the source code for mutual exclusion. SR further associates coherence actions with the synchronization primitives. For example, a contiguous portion of virtual memory, called a region, is associated with a critical section through the `BindRegion()` function call. Unlike traditional locks, the association between the lock data structure and the critical section is not merely a convention. SR requires that a region and a synchronization data structure, called a token, be bound at run-time, such that the token of one region cannot be used for a different region. The granularity of the region associated with a token is user-defined. A token can be bound to an entire multi-dimensional array or each row of an array can be bound to a different token to reduce the granularity of coherence. Before a shared region is accessed, either a `ReadAccess()` or a `WriteAccess()` synchronization must be issued. Normally, both a locking action and a data coherence action are associated with the request for access. `ReadAccess()` synchronization is intended for read-only data sharing and `WriteAccess()` is for read-write data sharing. At the end of the critical section, a `ReadDone()` or `WriteDone()` synchronization must be issued.

By exploiting information about the binding between token, region, and the type of access required for the critical section, the SR run-time system can enable caching and keep the caches coherent in software. The specific coherence algorithms are varied, but include the cache-validate algorithm, based on cache invalidation, and the cache-flush algorithm, based on cache copyback and invalidation. For the cache-validate approach, a state vector is associated with each region. When either a `ReadAccess()` or `WriteAccess()` is issued, the state of the region is checked and, at this time, it must be HV, for Has a Valid copy. Otherwise, the relevant cache lines are invalidated so that the next cache miss will be filled from main memory. For a `ReadDone()`, no action needs to be taken. However, for a `WriteDone()`, the contents of the cache are copied back to main memory and the state of the region for all the other processors must be set to be HI, for Has an Invalid copy. Thus, the cache-validate algorithm is analogous to hardware cache invalidation schemes, but it uses the knowledge of region bindings and access control points to maintain the state of the region and take appropriate coherence action in software.

The main advantage of the SR model is that it allows shared data to be cached on parallel architectures that do not provide hardware cache coherence. Hector and the TC2000 are two examples of multiprocessors that can benefit from the latency-reducing role of caches with the SR approach. The performance of the SR implementation on Hector is encouraging [Sandhu *et al.*, 1993]. Also, since SR is a software solution to the cache coherence problem, it is potentially less expensive than the design and manufacturing costs of hardware cache coherence. Furthermore, the SR software supports a programmer-defined unit of coherence and can support a variety of different coherence algorithms simply by extending the run-time system.

The disadvantages of the SR model include the need for explicit binding between regions and tokens, and the inability to detect when memory is incoherent due to a programming error. For the SR run-time system to work correctly, the user is responsible for explicitly binding a region of shared memory with a token. This is an extra responsibility and potentially error-prone process for the programmer. The programmer is also responsible for inserting the necessary `ReadAccess()` and `WriteAccess()` synchronizations around critical sections. Admittedly, the need for explicit synchronization is also a part of other shared-memory systems, but the need for explicit region binding is a new responsibility for the programmer. Also, when a programmer fails to correctly bind shared memory with tokens and when the synchronizations are not properly inserted, the application may not detect the coherence errors. The application continues to access old data from the local cache without any run-time error message. Shared-memory programs suffer from the safety problem associated with improper synchronization, but SR can also suffer from improper region binding. Ideally, the compiler or run-time system should automatically handle the binding and coherence actions.

In the context of distributed-memory platforms, the C Region Library (CRL) [Johnson *et al.*, 1995], is similar to the SR model, with the same disadvantage of requiring the programmer to explicitly bind memory regions to a region identifier.

2.2.4 Distributed Shared Memory: TreadMarks

A variety of distributed shared memory (DSM) systems have been designed, implemented, and evaluated [Stumm and Zhou, 1990]. Ivy [Li, 1988] was the first DSM system. Since then, other systems, including Midway [Bershad *et al.*, 1993] and Blizzard [Schoinas *et al.*, 1994], have also been described in the literature. The TreadMarks system [Keleher *et al.*, 1994,

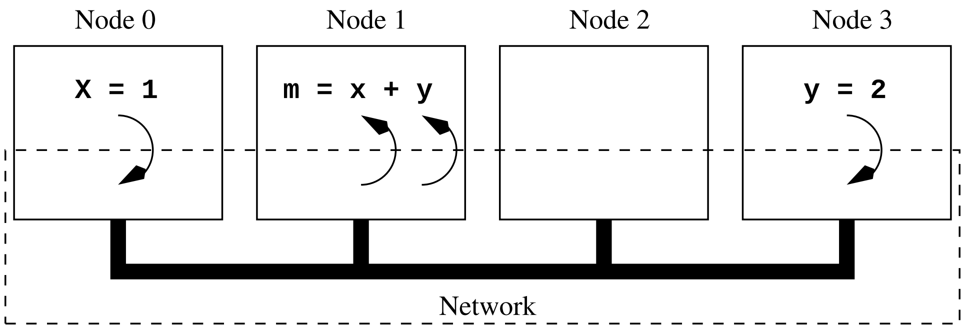


Figure 2.4: Distributed Shared Memory (DSM) Architecture

Amza *et al.*, 1996] is a follow-up to the work on Munin [Carter *et al.*, 1991]. What makes TreadMarks particularly interesting is that it is designed to work with a number of different implementations of Unix. Although some of its implementation strategies are similar to those of other DSM systems, it also has some unique features.

TreadMarks is implemented on top of the Unix operating system running on a NOW. Although these machines do not support shared memory in hardware, TreadMarks can create the illusion of shared virtual memory (Figure 2.4). A software-based shared-memory abstraction is built around portions of each processor's local memory and the network to hide the underlying network messages. Loads and stores to shared data are translated, as necessary, to network sends and receives to other processor nodes for the data. A variety of IP-based or ATM-based networks can be used as the interconnect with current versions of TreadMarks. As with other DSM systems, and in contrast to Shared Regions, a page of virtual memory is the fixed unit of granularity at which the system maintains data coherence. To the programmer, TreadMarks provides the standard non-segmented memory layout of Unix. Within the same address space, there are both private pages and pages that can be shared between independent processes on different workstations.

The basic implementation strategy of TreadMarks is to manipulate the protection bits of virtual pages, through the `mprotect()` system call, such that accesses to shared pages can be intercepted when necessary to maintain data coherence. Since the page-based mechanisms of virtual memory are used, the granularity at which the shared data is managed is also per-page. TreadMarks records the status (e.g., valid, invalid) of the different pages of shared data and if the local copy of a page is invalid, the run-time system is responsible for intervening and obtaining a valid copy before the load or store is allowed to complete. By removing the read and write permissions on a invalid page, a standard Unix `SIGSEGV` signal is generated on any attempt to access data in the page. By installing a signal handler for the access violations,

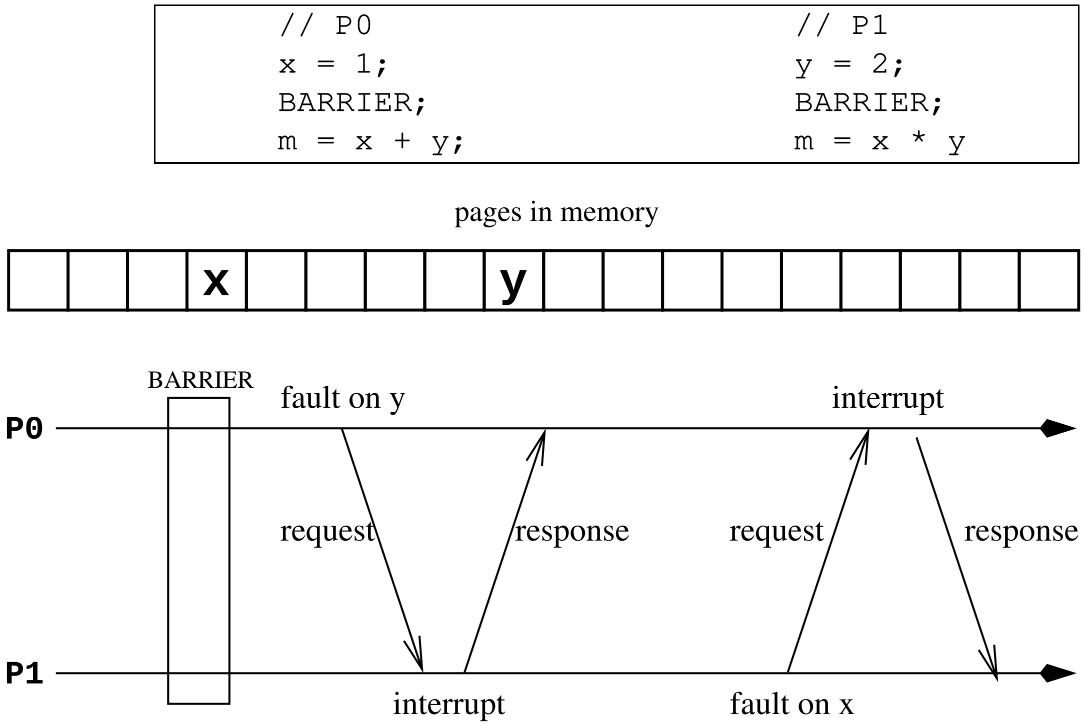


Figure 2.5: DSM Basic Mechanisms

the run-time system can determine the state of the page and take any required coherence action. Often, network communications and updates to TreadMarks-specific data structures are required. Aside from `mprotect()` and signal handling functions, the run-time system only uses unprivileged communication primitives, therefore TreadMarks is implemented entirely in user-level code.

Consider the simple example of two processes, `P0` and `P1`, executing the code shown in Figure 2.5. In the event graph, the timeline proceeds from left to right. After the synchronization barrier, the two processes attempt to read a shared value that was written by the other process before the barrier. Let us assume that the shared variables `x` and `y` are located on separate shared pages. Therefore, `P0` will incur a page fault when it tries to read `y` while evaluating the expression `m = x + y` and `P1` will incur a page fault when it tries to read `x` while evaluating the expression `m = x * y`. In the case of `P0`, the fault on `y` is handled by sending a request message to `P1`, interrupting `P1` to ask for the data, waiting for the response message with the required data, and then proceeding with the computation. The fault on `x` is handled in a similar manner. Note the characteristic request-response pattern of network messages.

Although TreadMarks supports the normal Unix memory layout, the memory consistency model presented to the programmer is based on lazy release consistency (LRC) [Keleher *et al.*, 1992]. As with release consistency (RC), LRC associates acquire and release synchronization properties with the normal locking and barrier synchronizations of the application. With RC, updates to shared data do not need to be communicated to remote processors until the release synchronization. However, with LRC, the communication of updates is delayed until the next acquire synchronization and only the acquiring process receives the updates. The main benefit of the lazy propagation of updates is that only the processor issuing the next acquire needs to receive the updates. Furthermore, if the next acquire is issued by the same processor that issued the last release, no communication to a remote processor is required. RC's strategy is better for pipelining the update communication operations, but LRC potentially requires fewer messages and data on the network. Since TreadMarks is intended for NOWs with general-purpose networks, the reduction in communication traffic may be an important tradeoff with the latency-hiding properties of RC.

Another unique aspect of TreadMarks is that, instead of communicating an entire page, a run-length encoded summary of the updates to the data, called a diff, is created and communicated. To create a diff, TreadMarks uses a copy, or twin, of each shared page before the first store operation and does a byte-by-byte comparison of the page with its twin. Maintaining twins and creating diffs can be computationally expensive, so diffs are only generated on demand through a process known as lazy diff creation. By using diffs, TreadMarks reduces the number of bytes that need to be communicated to maintain data coherence.

The diff data structure is also used to reduce the performance losses due to *false sharing* and having multiple writers to a shared page. The problem of false sharing occurs when two data items are located on the same virtual page and the updates to one data item do not need to be communicated to processors using the other data item. A store operation to one data item will unnecessarily invalidate the entire page on all other processors because of the large granularity of sharing. In general, false sharing can be eliminated if the data structures are located on different virtual pages or if the granularity of sharing in the DSM can be smaller than a page. However, that may not always be convenient or possible. TreadMarks deals with false sharing and multiple writers by allowing different processors to read and write to different parts of the same virtual page without necessarily invalidating the entire page. A multiple-writer protocol [Carter *et al.*, 1991] allows acquire operations on different processors to complete even if they will be accessing data on the same shared page. If the sharing is truly false or if the updates to memory are to different addresses, then the consistency of memory is maintained with fewer messages.

One advantage of DSM is that programmers can develop shared-memory applications on hardware that does not support shared memory. TreadMarks, in particular, uses standard Unix processes and memory protection functions, so it can be ported to a large number of platforms. The disadvantages of TreadMarks include the high cost of data and process synchronization and the memory overheads of maintaining diffs and other support data. For example, Unix signal handling is relatively slow because it is a general-purpose mechanism [Thekkath and Levy, 1994] that was not designed to support DSM. However, TreadMarks makes extensive use of Unix signals to implement its memory protection and shadowing techniques. Also, the need to maintain twins of shared pages and to create diffs can be expensive in terms of storage, copying, and comparison overheads. Therefore, TreadMarks provides a familiar shared-memory programming environment on platforms without special hardware support, but the software-only approach has performance and storage overhead costs.

2.2.5 Object-Based Shared Data: Orca

Orca is an object-based language and run-time system that supports communication between distributed processes via a shared-data object model [Bal *et al.*, 1992, Bal *et al.*, 1998]. A special compiler translates high-level constructs in the Orca language that access the shared data (e.g., assigning a value) into lower-level code that may result in messages across the network. The details of how the shared-data object is actually accessed is transparent to the programmer. The system has been implemented on top of a variety of distributed-memory platforms, including a NOW running Unix and a NOW running the Amoeba operating system [Mullender *et al.*, 1990]. Although the hardware does not provide a shared address space, Orca provides a shared-data abstraction via an encapsulation mechanism. The shared data can only be accessed through language constructs, which allows the compiler and run-time system to maintain mutual exclusion and coherence on the shared data.

Other shared-data object-oriented parallel programming systems also exist, including SAM [Scales and Lam, 1994], Mentat [Grimshaw *et al.*, 1996], and ABC++ [O'Farrell *et al.*, 1996]. Although COOL (discussed previously) and μC++ [Buhr *et al.*, 1992] are designed for shared-memory platforms, they do use the encapsulation features of C++ (and extensions) to control the access of shared data through a monitor construct. However, Orca is particularly interesting because it is designed for distributed memory and because of the tight integration of a new parallel language with a compiler and run-time system.

The syntax of Orca resembles that of Modula-2 with parallel programming extensions. A process is defined with the keyword `process`, instead of `procedure`. Processes are explicitly created using the `fork` keyword. Shared-data objects are defined with the keyword `object`, similar to a record definition, and data objects are shared between processes only if the object is a parameter to the process when it is forked.

Orca's concept of an object has similarities to both the object-oriented notion of an object and the monitor concurrency-control mechanism. However, since Orca's shared-objects do not support inheritance or polymorphism, it is most accurately described as an object-based system. As with a monitor, Orca's shared objects can only be accessed via procedures and functions, called operations, encapsulated within the object definition. Mutual exclusion is implicit with each invocation of an object's encapsulated operation. Synchronization within a shared object is achieved with guard conditions that block the executing process until the boolean conditions of the guard are satisfied. No other granularity of mutual exclusion and no other synchronization mechanism is available.

Orca's notion of guards is similar to the notion of an automatic-signal monitor [Buhr *et al.*, 1995]. However, since the classical notion of a monitor implies a central resource and no support for replication, Orca's objects are fundamentally different from monitors in their implementation. Orca's objects are abstract data types with support for mutual exclusion and declarative guard-based synchronization.

Two key advantages of the Orca approach are its safety and convenience. Implicit mutual exclusion and declarative synchronization enhance the safety of the system by making it impossible to violate a critical section and by simplifying the semantics of conditional synchronization. The programmer does not need to insert acquire, release, `WriteAccess()`, or `WriteDone()` synchronization primitives into the source code to maintain correctness when accessing shared data. Also, the programmer can reason about the synchronization behaviour of a shared-data object by inspecting the static guard conditions instead of having to consider the run-time behaviour of a traditional monitor's signal and wait operations [Buhr *et al.*, 1995]. Furthermore, the shared-data object model automatically maintains data coherence while transparently replicating and migrating shared data as needed to improve the locality of memory accesses. The automatic mutual exclusion, synchronization, and data coherence of the shared-data object model are convenient and attractive to the application programmer.

The disadvantages of the Orca approach include its non-mainstream programming language, its use of sequential consistency, and its reliance on broadcasting for the data coherence protocol. First, despite the advantages of Modula-2 as a research and teaching language, it is not commonly used by the parallel programming community. Of course, many of the ideas in Orca may be applicable to other languages, but Orca itself is purely a research language. Second, Orca's update protocol for shared objects is based on sequential consistency, which can suffer from the performance problems that are addressed by relaxed consistency models. Last, the early implementations of Orca depend on a reliable broadcast mechanism to implement the write update-based coherence protocol. With some NUMA architectures, such as meshes, a scalable and efficient broadcast mechanism may not exist. Of course, other data coherence protocols may potentially be incorporated into Orca's run-time system.

2.2.6 Template-Based Process Structures: Enterprise

Unlike explicit message passing and shared data, Enterprise uses a restricted function call interface to control the interaction of parallel processes [Schaeffer *et al.*, 1993]. In contrast to the systems surveyed so far, Enterprise is based more on a process model than a memory model. Conceptually, processes implicitly communicate through the arguments of a function call, instead of through explicit messages or explicit changes to shared data. In fact, Enterprise emphasizes the use of pure functions, which have no side effects, so that the system is free to pipeline, multiplex, and redirect a function call to a process according to higher-level abstractions, such as process templates. Different templates can be attached to different functions and templates can be hierarchically nested. Templates can also be changed without any changes to the user's source code; Enterprise simply changes the system-provided code that implements the template.

Enterprise uses a top-down approach to specifying how different sequential processes communicate to create a parallel application. The individual processes are programmed and debugged as sequential code and then the parallelism is specified by attaching predefined templates to the application. A sequential procedure call is transformed into a message to a parallel process with the arguments of the call. A return statement becomes a message back to the caller with the result of the computation. Calls to parallel processes are non-blocking and the return results are bound to the function calls using futures [Halstead, 1985]. Enterprise templates are provided by the system and have been designed so that they can be hierarchically applied to exploit nested parallelism. For example, a "master and workers" template creates parallelism through the replication of many worker processes executing pure functions. Pipelining is another strategy for creating parallelism, so a "pipeline" template overlaps the computation of the different functions in a call sequence. Consequently, a typical Enterprise application consists of user-supplied functions, user-specified call sequences, and a system-supplied set of process templates called assets.

The main advantage of the template-based approach, and related work on skeletons [Cole, 1989] and frameworks [Singh *et al.*, 1991], is the ease with which a programmer can create correct parallel programs. Parallel communication structures, such as master-worker, a pipeline of processes, and a dynamic group of processes (i.e., the number of processes in the group can change during execution), are common enough to be considered idioms and can be predefined as system-provided templates. In Enterprise terminology, these process idioms are the department, line, and contract assets, respectively. The programmer supplies the code for the master and worker processes, but Enterprise provides the code to implement the master-worker relationship. The implementation and validation of the template are the responsibility of the developers of Enterprise. Furthermore, the semantics of attaching a template to a call graph allows different templates to be interchanged and templates to be nested without requiring changes to the user's code. Therefore, the user may experiment with process groups, pipelines, or groups-of-pipelines without changing the source code. Enterprise inserts the correct code to implement the template thus allowing the programmer to add and experiment with parallelism with safety. From a pragmatic point of view, the ability to safely experiment with different configurations and structures is very valuable.

Since Enterprise processes communicate only through the parameters of a procedure call, the programmer sees a distributed-memory model where processes communicate through implicit messages. As with Orca and TreadMarks, the Enterprise system is targeted for networks of workstations. Other systems focus on how to provide consistent memory semantics, but Enterprise focuses on how to provide correct process interaction semantics. The Enterprise model demonstrates the value of being able to safely change process interactions without requiring changes to the user's code.

The disadvantages of the Enterprise system include the inability to pass pointer-based data structures, such as graphs, in parallel function calls and the difficulty of hand-optimizing and customizing an Enterprise application. First, since Enterprise assumes a distributed-memory architecture and since it requires a stateless function call interface between parallel processes, pointer-based data structures are not allowed as parameters. Last, Enterprise is a high-level programming system that does not provide an easy way to implement abstractions that are not already supported by the system. For example, there is currently no method for a user to define a new process template or to modify the behaviour of an existing one. Applications that do not fit well with the existing templates will not parallelize well under Enterprise.

2.3 Data-Sharing Idioms and Optimizations

In designing computer systems, it is natural to identify and optimize for commonly occurring idioms. Therefore, in addition to general-purpose mechanisms, it is important for a system to easily and seamlessly support any commonly recurring structures and patterns. For example, researchers have identified a number of data-sharing patterns that re-appear in various distributed systems and applications. Data-sharing idioms and patterns such as read-

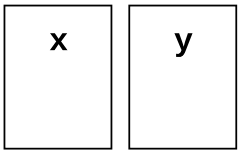
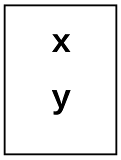
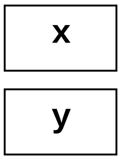
	DSM	**DSD**
What is shared?	Memory in an address space	Data structures
Naming	Pointer	Abstract Data Type and pointer
Key mechanisms	Page faults for virtual memory	Abstract Data Type and programmer's interface
Unit of management	Fixed: page  or 	Variable: object or region 
Unit of sharing policy	All shared pages; sometimes per-page	Object or region
Can suffer from false sharing?	Yes	No
Can alter sharing policy on per-context basis?	Possible, but still subject to false sharing	Yes

Table 2.1: Distributed Shared Memory (DSM) vs. Distributed Shared Data (DSD)

intensive, write-intensive, producer-consumer, migratory sharing, and others have previously been noted [Bennett *et al.*, 1990b, Bennett *et al.*, 1990a, Falsafi *et al.*, 1994]. Consequently, various compiler and run-time techniques to optimize performance for common data-sharing idioms have been proposed.

Parallel programming systems for distributed-memory architectures differ widely in their ability to optimize for specific data-sharing idioms. Enterprise is aimed at supporting common process structures, therefore it is relatively limited in its support for data-sharing structures. Message-passing systems such as MPI are low-level enough that they can implement almost any idiom given sufficient programmer effort. However, given the explicit nature of message passing, small differences in the idiom may require many changes to the source code in order to correctly match the sends and receives. Of the two remaining distributed systems already discussed, TreadMarks and Orca, their ability to support data-sharing idioms is greatly influenced by their original design (Table 2.1). As a page-based DSM system, TreadMarks manages and selects policies on a per-page basis. As an object-based system, Orca manages and (can potentially) select policies on a per-object basis. We introduce the term *distributed shared data* (DSD) to refer to systems, like Orca, that offer a variable granularity of sharing and policy selection. One particular approach to implementing a DSD system is to develop abstract data types (ADT) for shared data. In the following discussion, we argue for the benefits of the DSD approach and discuss some of the issues behind the design of the Aurora system, which is based on ideas from both distributed shared data and abstract data types.

Ideally, the parallel programming system should allow the data-sharing behaviour of each unit of shared data (or object) to be optimized independently of other units of shared data. 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 [Lu *et al.*, 1995, Lu *et al.*, 1997]. If shared variables `x` and `y` are co-located on the same page (Table 2.1), updates to `x` will change the cache state of the entire page, including the state of variable `y`. Even if the value of `y` never changes, there may be overheads associated with reading its value because of the interference of updates to `x`. One possible solution to this problem is to explicitly place variables `x` and `y` on different pages so that updates to the page with variable `x` do not change the state of other variables. Of course, dedicating an entire page of memory for a single variable causes memory fragmentation, so it is not a general-purpose solution.

In DSD systems, false sharing is avoided by managing independent data as separate objects, or regions of memory, with their own state information. Variables `x` and `y` are managed as separate units of data and updates to one variable do not impact the state of the other variable (Table 2.1). Consequently, it is also easy to select a different data-sharing policy (i.e., when and how to communicate data) for different shared-data objects. Some DSM systems do support per-page data-sharing policies, but all of the shared data on a page must still share the same policy. If for one phase of the computation it is reasonable for two shared variables to share the same policy, but for another phase the two variables require different policies, a page-based DSM system would experience interference between the two variables.

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

```
// 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 );
```

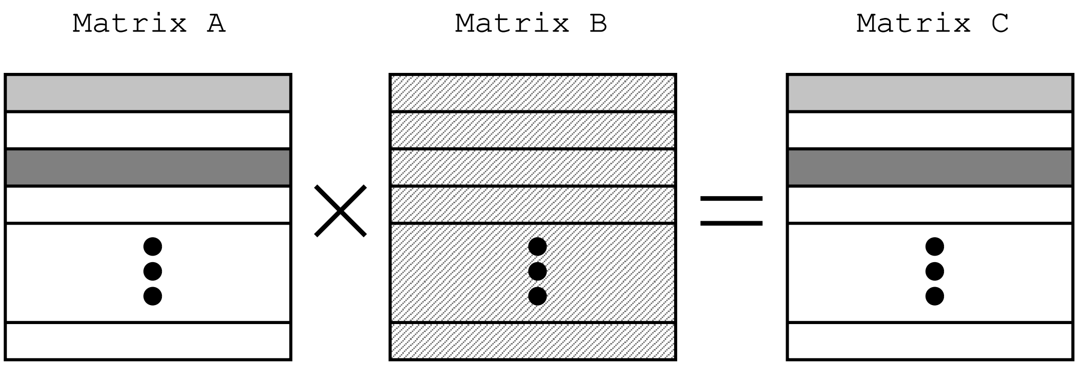


Figure 2.6: Matrix Multiplication: Different Access Patterns

Let us consider a concrete problem that exhibits different data-access patterns and can benefit from the flexibility of the DSD approach. In basic dense matrix multiplication, without blocking or other sophisticated enhancements, three matrices are accessed during the computation (Figure 2.6). Each matrix has a different access pattern and different properties.

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.

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.

A typical parallelization strategy for matrix multiplication is to partition the independent rows of Matrix A. Each process multiplies its rows of Matrix A with all of Matrix B to compute the rows of Matrix C. Consequently, all processes need to access all of the elements of Matrix B, resulting in a large working set for Matrix B. At the start of the computation, the data for all three matrices may be distributed across all the nodes involved in the computation. Thus, the challenge is to move and share the distributed data in the most efficient way possible.

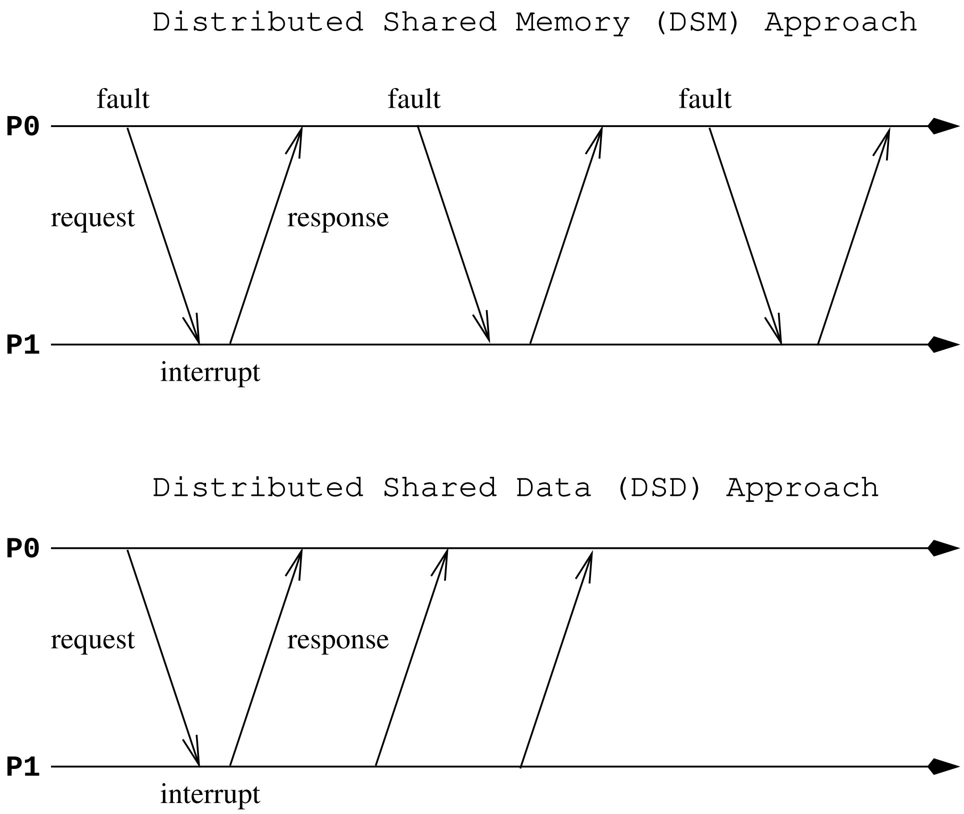


Figure 2.7: Optimizing Patterns: Bulk-Data Transfer

In TreadMarks, all shared data are located on the same pool of shared pages and all of the pages are managed according to the same policy. Therefore, the differences in how the three matrices are used cannot be exploited to improve performance. In contrast, a DSD system (but not Orca specifically) can attempt to provide data-sharing policies to match the needs of each matrix. We consider specific optimizations for two of the matrices: Matrix B and Matrix C.

Consider Figure 2.7 and the pattern of network messages required to move a large amount of data from process `P1` to process `P0`. Recall that all processes require a complete copy of Matrix B, which can be megabytes and hundreds of pages of data. The top event timeline shows a typical DSM system and the bottom timeline shows a potential optimization in a DSD system. Instead of faulting on each page associated with Matrix B and incurring a request-response message pair for each page, it is more efficient to perform a bulk-data transfer and send a single request for all of the contents of Matrix B. After the initial request, there is no need for further requests. The multiple response messages in the DSD case reflect the need to packetize the data for the network.

However, in order to do a bulk-data transfer, the parallel programming system must be able to identify the extent of the data to be moved. The system wants to transfer all the data for Matrix B and it wants to avoid transferring data that is not part of that particular data structure. In a DSM system, pages of shared data are not generally associated with a given data structure. All pages belong to the same pool of pages with no distinction between which pages belong to which high-level data structure. It is possible to maintain this information in a DSM system, which would be equivalent to the region binding and access control primitives of SR (Section 2.2.3), but it is not typically done and it is not done in the TreadMarks system. In a DSD system, Matrix B is a single unit of sharing that is separate from Matrix A, so it is easy to determine the data that needs to be transferred. If Matrix B implemented as an ADT, bulk-data transfer can be part of the interface to the ADT.

The benefit of using a bulk-data transfer is a significant reduction in the number of messages required to move the data and better tolerance for network latencies. As can be seen from the timelines, almost all of the request messages can be eliminated by using a bulk-data transfer. Also, and perhaps most importantly, there is no extra latency as `P0` waits for each response message. After the latency of waiting for the first response, the remaining response messages arrive in pipeline fashion. A fundamental performance problem with request-response messages is that the requesting process is (often) idle while waiting for the responding process to return the data. Latency is incurred in network for the outgoing message, at the remote process's node while handling the request, and (again) in the network for the response message. Bulk-data transfer avoids most of these latencies by eliminating the need for multiple request messages and by pipelining the data transfer.

A drawback of bulk-data transfer is a potential increase in contention for the network and the various hardware and software layers involved in the data transfer. The request-response message pattern of the DSM-based system acts as a control flow mechanism that can reduce the amount of contention in the system. Since bulk-data transfer is more aggressive in how data is streamed between sender and receiver, multiple senders and multiple receivers may actually interfere (i.e., contend) with each other to a greater extent than without bulk-data transfer. A performance trade-off is made between the extra messages (and latency) of request-response and the drawbacks of (possibly) more contention with bulk-data transfer. If contention is a problem with a specific bulk-data transfer protocol, then a form of control flow may be required.

In the case of Matrix C, the fact that the matrix elements are write-only can be exploited to improve performance. Consider the case where `P0` is writing to Matrix C, but `P1` contains the most up-to-date values for the matrix (Figure 2.8). In a DSM system, there is a separate fault and request-response message pair for each page of Matrix C that is updated, as shown

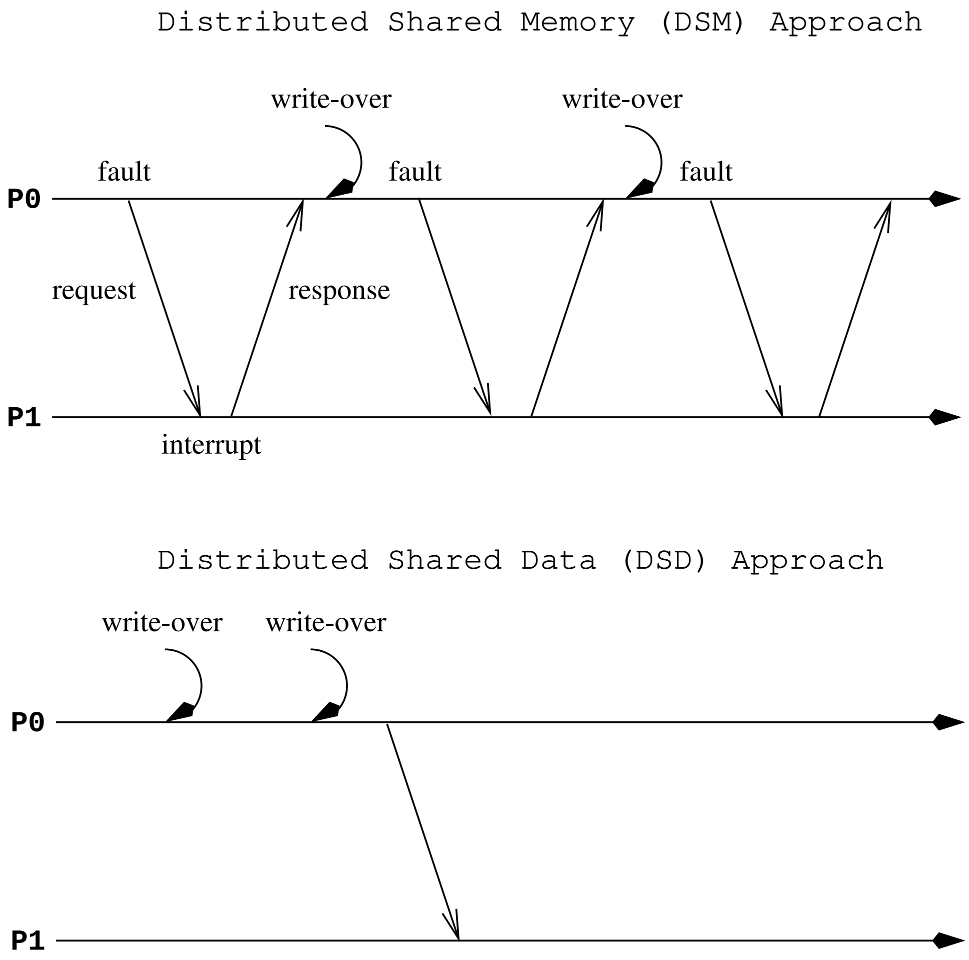


Figure 2.8: Optimizing Patterns: Write-Only Data

in the top event timeline. Even though the current contents of Matrix C will be immediately written-over, TreadMarks and similar systems must obtain the most recent values from `P1` before the write is allowed to continue. There are two reasons for this behaviour. First, there is no mechanism to inform TreadMarks that Matrix C is a write-only data structure, therefore TreadMarks assumes that any shared data that is accessed can potentially be accessed for both reads and writes. Again, the access control primitives from SR may be applicable to DSM systems, but it is not currently supported in TreadMarks. Second, as an inherent part of its design, TreadMarks must keep track of the current values in Matrix C for its cache coherence protocol.

In a DSD system, access annotations can be easily supported such that many of the network messages can be avoided for write-only data structures. In a sense, the shared-data ADT is analogous to a bound region in the SR system. Therefore, if Matrix C is write-only, process

`P0` does not have to obtain the most up-to-date values before overwriting them. Each update to Matrix C is recorded and at the end of the computation, the new values can be communicated to process `P1`, or to any other process, with a minimum number of messages. The benefits of this optimization include the elimination of request messages and their latencies and, when compared to TreadMarks, the elimination of protocol processing overheads such as twinning and diffing.

2.4 Summary

We have examined a number of parallel programming systems that differ in their basic programming mechanisms and in the level and type of abstractions provided to the programmer. A brief summary of the key advantages and disadvantages of these systems can be found in Table 2.2. Message-passing systems, such as MPI, provide a low-level and explicit message interface. They can be fast, but they can also be error-prone to work with. Hardware-based shared memory and DSM systems, such as DASH, SR, and TreadMarks, are based on a low-level load-store interface. A load or a store operation is all that is required to access the data. Object-based shared-data systems, such as Orca, have a higher-level interface and automate both synchronization and coherence actions. Shared data in Orca can only be accessed by functions encapsulated with the shared data, so the level of abstraction and the interface is higher. In contrast to the other systems, Enterprise focuses on a structured approach to process management. Process structures can be nested hierarchically and changed without any changes to the source code. Enterprise's stateless function call interface limits the interaction between parallel processes and makes the high-level template-based approach possible.

An important point of comparison for parallel programming systems is their ability to support commonly recurring structures and patterns. Data-sharing idioms have been identified in a number of applications and performance can often be improved by exploiting the semantics of these idioms. If it is known that a large amount of data must be transferred between processors, then an appropriate bulk-data transfer protocol can significantly reduce the number of network messages. And, if it is known that data is write-only, various data communication and protocol overheads can be reduced. Consequently, flexibility and support for idioms is an important design goal in the Aurora system.

System	Advantages	Disadvantages
Message Passing		
MPI Distributed memory.	Portability and performance.	Low level of abstraction may require substantial changes to data structures and source code to alter data-sharing and data-movement policies.
Shared Memory: Load-Store Access Interface		
DASH/COOL NUMA shared memory and cache coherence in hardware.	DASH's hardware-based cache coherence can achieve high performance. COOL has facility to specify locality with safety.	DASH only supports cache line granularity of coherence and only supports one coherence protocol. COOL does not support alternate coherence protocols through software.
Shared Regions (SR) NUMA shared memory in hardware. Cache coherence in software.	Enables caching of shared data with minimal hardware support. Supports variable granularity of coherence and a variety of coherence protocols.	Requires convention of explicit binding of regions and use of access control primitives. Cannot detect when convention has been violated and data is incoherent.
TreadMarks Distributed shared memory on Unix workstations through software.	Requires minimal hardware and software support. Provides familiar Unix-style memory layout. Supports multiple writer protocol and lazy release consistency.	Storage and computation overheads for twins and diffs. Fixed size, large granularity of sharing requires complex and expensive solutions to support multiple writers and reduce communication overheads.
Object-Based Shared Data: High-level Access Interface		
Orca Distributed memory. Data sharing and coherence through software.	Requires minimal hardware support. Encapsulation of shared-data object improves integration and safety. Object-level granularity of coherence helps to avoid false sharing. Uses replication to improve locality.	Non-mainstream language. Based on sequential consistency. Current coherence protocol relies on scalable broadcast mechanism, which is not always available.
Restricted Function Call Interface		
Enterprise Distributed memory. Template-based hierarchical process structures.	Easy to build and experiment with different hierarchical process structures with safety (including dynamic process structures).	Stateless function call interface limits interaction between parallel processes. Difficult to hand-optimize system-provided templates.

Table 2.2: Summary of Parallel Programming Systems

Chapter 3

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. The process models abstract the hardware processors and the data-sharing model abstracts the various forms of data communications. Aurora is an object-oriented and layered system (Table 3.1) 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 process and 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).

Layer	Main Components and Functionality
Programmer's Interface	Process models (`For_all()`, teams of threads, active objects) Distributed vector and scalar objects *Scoped behaviour*
Shared-Data Class Library	Handle-body shared-data objects Scoped handles implementing data-sharing optimizations
Run-Time System	Active objects and remote method invocation (currently, ABC++) Threads (currently, POSIX threads) Communication mechanisms (shared memory, MPI, UDP sockets)

Table 3.1: Layered View of Aurora

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.

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. 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. In Chapter 4, we describe the implementation details of Aurora, but for now, we focus on the application programmer's point of view.

3. To the implementor of the run-time system, scoped behaviour is a mechanism for specifying high-level semantic information about how shared data is used by the parallel program. As we will see in Chapter 4 and Chapter 5, scoped behaviour's ability to carry semantic information across the software layers can be exploited to optimize data communications.

In addition to scoped behaviour, Aurora is notable because it supports more than one process model and different process models can be mixed within the same program. The details are discussed below.

Aurora provides a shared data programming model on distributed-memory architectures through C++ classes that implement shared-data objects as abstract data types (ADT). Using the ADT interface, scalar and vector data can be distributed across, and accessed from, different address spaces. Also, the ADT interface can be changed and optimized using scoped behaviour in order to improve the program's performance.

Before we can discuss the implementation details of Chapter 4 and the parallel programs of Chapter 5, we must introduce the process models and shared-data objects of the Aurora system. Then, we discuss how the various abstractions and mechanisms are used for a simple loop example and in a parallel matrix multiplication example. The emphasis throughout the chapter will be on the Aurora's API and how the abstractions are used in practice.

3.1 Process Models

A parallel application consists of a set of multiple processes or threads. Classic processes are address spaces with a single thread of control. Since a distributed system consists of a set of different address spaces on different processor nodes, the overall design of how the different processes interact is often called the *process model*. However, contemporary operating systems support the more general abstraction of an address space with multiple threads of control. Therefore, a parallel application on a contemporary distributed system can consist of multiple processes and multiple threads.

Sometimes, as is the case with Aurora, different threads are used to implement a multi-threaded run-time system but there is only one thread used for the actual computation. From the point of view of the application programmer, there is only one thread of control in the user's source code. Therefore, for simplicity, the term "process model" will refer to a set of processes, but includes the possibility of multiple threads in each address space. Conversely, the term "thread" also includes the special case of single-threaded "processes."

More importantly, different operating systems and system libraries can vary greatly in the specific ways in which processes and threads are managed. The low-level details of how processes and threads are created, bound to processors, identified, synchronized, and destroyed can be overwhelming. Therefore, many programming systems provide high-level process models that simplify the task of programming and controlling processes.

A variety of process models have been described in the literature [Wilson, 1995]. An important characteristic of a process model is whether the individual threads execute the same function during a given computational phase, or whether different threads are allowed to execute different functions (columns of Table 3.2). Of the two possibilities, *task parallelism* is the most general-purpose model since it allows each thread in the application to execute a different function. Each thread can be independent of the other threads. Of course, the threads can be constrained such that they execute the same function. *Data parallelism* can be viewed as a restricted form of task parallelism in which each thread computes the same function, but on different portions of the same data structure. Typically, data parallelism implies that there is

	Same Function	Different Functions
Same Data Structure, Different Elements	Data parallelism provided by Aurora's `For_all()` mechanism	Task parallelism provided by ABC++'s active objects
Unrestricted Data Structure(s)	Single program, multiple data (SPMD) provided by Aurora's teams of threads	Task parallelism provided by ABC++'s active objects

Table 3.2: Process Models in Aurora

some synchronization, either tight or loose, as each thread progresses through the computation.

In the *single program, multiple data (SPMD)* process model, a team of independent threads execute the same program and, usually, the same function. Since the threads are not bound to a single data structure, SPMD can be viewed as a generalization of data parallelism (Table 3.2). The SPMD model can also be viewed as a more restrictive form of task parallelism in terms of the functions that can be executed by the threads.

In theory, different process models can be used within a single application. For different phases of a computation, a particular model may be more appropriate than another model. However, in practice, a single model is often used throughout an application in order to simplify the implementation.

The research goals of this work relate primarily to the data-sharing model, therefore the process models in Aurora are drawn from existing systems (Table 3.2). Aurora's mixed-parallelism process model includes task parallelism, data parallelism, and SPMD. Furthermore, the various process models can be mixed within a single application in reasonable ways. Although some other existing systems also support mixed-parallelism process models [Bal and Haines, 1998], it is still not a capability that is common to all programming systems. As we will see in Chapter 5, different applications may require different process models.

First, task parallelism is provided by the underlying ABC++ library through *active objects* [O'Farrell *et al.*, 1996]. Active objects are normal C++ objects that contain their own thread of control. Figure 3.1 illustrates how active objects are defined, instantiated, and invoked in ABC++. The ABC++ library provides class `Pabc`, which is the active object base class. It is a standard C++ class that, as part of its instantiation (i.e., function `Pabc_create()`), creates an independent thread of control that is bound to the new object. The thread is generally used to execute method invocations (e.g., `Pvoid()`) on the object, but it can execute any arbitrary

```
#include "aurora.H"                              // Also includes ABC++.H

// New user-defined class for an active object
class MyDerivedClass : public Pabc {            // Class Pabc is from ABC++
    ...
    void someMethod() { ... };
    ...
};

int main( ... )
{

  // Active objects are pointed to, created, and accessed using ABC++

  Pabc_pointer<MyDerivedClass> aoPointer;    // Pointer to active object

  // Instantiate the active object and bind object + thread to pointer
  Pabc_create( aoPointer );

  // Method invocation:  handled by object's thread
  Pvoid( aoPointer, MyDerivedClass::someMethod );
} // main
```

Figure 3.1: Example: Active Objects in ABC++

code associated with the object. The programmer may define a new class derived from the active object base class (e.g., class `MyDerivedClass`), such that objects of the new class also have a thread of control. In this way, active objects support an object-oriented form of task parallelism.

Second, data parallelism in Aurora is provided by a simple `For_all()` construct that invokes the same method, in parallel, on all the elements of a given data structure. For example, if `SharedVector` is a vector object with shared data, then `For_all( SharedVector, computeA )` invokes method `computeA()` of `SharedVector`'s class for all the vector elements. Currently, no Aurora application makes use of this particular process model, therefore we do not discuss the `For_all()` mechanism further.

Last, Aurora provides a fairly standard SPMD process model that is similar to the PCP system [Brooks III, 1988]. 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. Due to its overall flexibility and simplicity, the SPMD model is used for all the applications discussed in this dissertation. The details of the SPMD process model are illustrated below.

3.2 Shared-Data Objects

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. 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 is local or remote. Therefore, as with DSM systems, Aurora provides the illusion that local and remote data are accessed using the same mechanisms and syntax. In reality, Aurora uses an ADT to create a shared-data abstraction.

By default, shared data is read from and written to synchronously, even if the data is on a remote processor node. Although synchronous accesses can be slow, that data-access behaviour has the least error-prone semantics. For performance-critical data accesses, the programmer can specify different data-access behaviours. The examples below (Section 3.3 and Section 3.4) will demonstrate how this is done.

Aurora's data model requires that shared scalar and vector objects be created using Aurora's C++ class templates `GScalar` and `GVector`. For example:

```
// In C, the following is analogous to:
//   int scalar1; VNode scalar2;
GScalar<int>    scalar1;
GScalar<VNode>  scalar2;           // User-defined type VNode

// In C, the following is analogous to:
//   int vector1[ 1024 ]; float vector2[ vsize ];
//   VNode vector3[ vsize ];
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. Of course, accessing `scalar1` from a remote processor node will require network messages to be sent, but the details are handled by Aurora. Any of the C++ built-in types or any user-defined concrete type [Coplien, 1992] can be an independent unit of sharing, as with datatype `VNode` in `scalar2`'s definition.

Shared vectors, such as `vector1`, are similar to shared scalars, except that the vector elements can be distributed across processor nodes and the subscript operator is overloaded to support remote and local indexing. As with shared scalar objects, any concrete type can be used as the type of the vector elements. Parameters to the constructor specify the size of the vector and (optionally) data placement on the processor nodes. By default, if only the size of the vector is specified, as with `vector1`, `vector2`, and `vector3`, the elements are block distributed across all nodes. But, data placement can be controlled, as in the following examples:

```
// 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 ) );
```

In the example, `vector4` has 1024 elements block distributed across home nodes 0 and 1, with each block containing exactly 512 elements. Specifically, vector elements 0 to 511 are on processor node 0 and vector elements 512 to 1023 are on processor node 1. `vector5` has `vsize` elements all placed on processor node 0 of the computation (i.e., stored as a single block). The class `Nodes` is provided by Aurora and is used by the programmer to list the logical processor nodes for data placement during object construction. Normally, the nodes are numbered from 0 to $n-1$ for n physical nodes. The mapping of logical nodes to physical nodes is handled by the run-time system. Using `Nodes`, the programmer can explicitly list an arbitrary subset of all the processor nodes on which data is to be placed. If desired, all of the processor nodes in the current system can be listed using `Nodes`.

Finally, since it is a common data structure, Aurora supports shared two-dimensional matrices. For example,

```
// 512 x 512 matrix; block distributed across the nodes
// In C, the following is analogous to:
//   int vector6[ 512 ][ 512 ];
GVector<int>    vector6( MatrixDim( 512, 512 ) );
```

defines a two-dimensional matrix with 512 columns and 512 rows of integers. Class `MatrixDim` is provided by Aurora and allows for a convenient syntax to specify the sizes of the different dimensions.

(a) Original Loop	**(b) Optimized Loop Using Scoped Behaviour**
`GVector<int> vector1( 1024 );`	`GVector<int> vector1( 1024 );`
	{ *// Begin new language scope*
	`NewBehaviour( vector1, GVReleaseC, int );`
`for( int i = 0; i < 1024; i++ )`	`for( int i = 0; i < 1024; i++ )`
`vector1[ i ] = someFunc( i );`	`vector1[ i ] = someFunc( i );`
	} *// End scope*

Figure 3.2: Applying a Data-Sharing Optimization Using Scoped Behaviour

3.3 Example: A Simple Loop

Figure 3.2(a) demonstrates how a distributed vector object is instantiated and accessed. As discussed, `GVector` is a C++ class template provided by Aurora and `vector1` is a vector object with 1024 integer elements that are block distributed.

The programmer can assign values to the elements of `vector1` using the same syntax as with any C++ array. The overloaded subscript operator (i.e., `operator[]`) is an access method that determines whether the update to `vector1` at index `i` is local or remote. If the data is local, a write is simply a store to local memory. If the data is remote, a write results in a network message. Similarly, a read access is either a load from local memory or a network message to get remote data.

Implementation details will be discussed in Chapter 4, but we briefly sketch the main ideas at this time in order 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 semantically 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 acknowledgement. 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 acknowledgement 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 acknowledgement, 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 (Figure 3.2(b)) 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 acknowledgement 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 (Figure 3.2(b)): 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.

(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
```

Figure 3.3: Matrix Multiplication in Aurora

This particular scoped behavior is based on the release consistent memory model for writes [Adve and Gharachorloo, 1996]. 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. This refinement in Aurora's implementation of release consistency, where only the affected data objects are made consistent at the end of a scope instead of an entire page or all of memory, is referred to as *per-object release consistency*. In this way, Aurora partitions the consistency namespace to avoid unnecessary updates of shared data.

3.4 Example: Matrix Multiplication

We now consider a more complex example involving multiple shared-data objects and different scoped behaviour optimizations, namely that of non-blocked, dense matrix multiplication, as

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.

Table 3.3: Some Scoped Behaviours

shown in Figure 3.3. The basic process model is that of teams of threads operating on shared data in SPMD fashion. The preamble is common to both the sequential and parallel codes (Figure 3.3(a)). The basic algorithm consists of three nested loops, where the innermost loop computes a dot product and can be factored into a separate C-style function.[1]

The simplest way to create an SPMD team of threads is to instantiate class `Team`, which is provided by Aurora. The application programmer can optionally place the individual threads on the processor nodes in round-robin fashion, by explicitly listing the processor nodes, or by co-locating the threads with an existing shared-data object. Consider these illustrative examples:

```
// Team of 4 threads; round-robin node placement
Team    teamA( 4 );

// Team on Nodes 0, 1, and 2 (i.e., 3 threads)
Team    teamB( Nodes( 0, 1, 2 ) );

// Team co-located with GVector vector4
//   (i.e., one thread on Node 0 and one thread on Node 1)
Team    teamC( vector4.nodes() );
```

Conceptually, we can view an optimization as a change in the type of the shared object for the lifetime of the scope. As an example of per-object flexibility, three different data-sharing optimizations (Table 3.3) are applied to the sequential code in Figure 3.3(b) to create the parallel code in Figure 3.3(c). In later chapters, we will discuss the other scoped behaviours in Table 3.3.

[1]Figure 3.3(b) and Figure 3.3(c) are shown using static matrix sizes (e.g., `int mA[ 512 ][ 512 ]` and `GVector<int> mA( MatrixDim( 512, 512 ) )`, where class `MatrixDim` is provided by Aurora), in order to simplify the presentation. Dynamic matrix sizes that are not known until run-time are supported via an appropriate indexing function, as in sequential C/C++.

In the discussion below, the data-sharing patterns discussed in Section 2.3 motivate the choice of data-sharing optimizations. 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. 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 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.[2]

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 formal 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 (Section 2.3). Typical demand-paged DSM systems do not exploit knowledge about how the data is 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 is 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 is used.

[2]The `myTeam` parameter to `doParallel()` is a convenient way to specify which process team's threads are about to iterate over the data in case there are multiple teams. But, for all of the examples to be discussed, there is only one SPMD team of threads.

3.5 Experimenting with Optimizations

Aurora does not automatically parallelize an application. The typical methodology for porting applications to Aurora "by hand" requires that the programmer understands the parallelism in the application. There are three main steps. First, shared arrays and shared scalars are converted to `GVectors` and `GScalars`. The default synchronous access policy can be slow, but its performance can be optimized after the program has been fully debugged.

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

Lastly, various data-sharing optimizations can be tried on different bottlenecks in the program and on different shared-data objects. Often, the only required changes are a new language scope and a `NewBehaviour` macro. Sometimes, straightforward changes to the looping parameters are needed for owners-computes. For example, in the matrix multiplication program, owner-computes can be applied to vector `mC` instead, with read caches used for both vector `mA` and vector `mB`. The `dotProd()` function and the data access source code remain unchanged. The new optimization strategy uses more resources for read caches than the original strategy, but since `mC` is being updated, it is perhaps a more conventional application of owner-computes. Another optimization strategy would be to use owner-computes for both vector `mA` and vector `mC`, and a read cache for vector `mB`. Reverting back to the original strategy is relatively easy. For the application programmer, the ability to experiment with different optimizations by making only limited code changes, can be valuable.

In contrast, a message-passing system typically requires many error-prone code changes in order to experiment with different parallelization and data-sharing strategies. Changing the parameters or order of an explicit message send also requires a change in the corresponding explicit message receive. If the data exchange is complex, say in a loop or within nested if-else control flow structures, it is easy to make a mistake or overlook a required change in the source code. It is easy for a programmer to mismatch the sends and receives, resulting in poor performance in the best case and a deadlocked program in the worst case. A message-passing system has great expressive power to orchestrate data movement and synchronization, but that power comes at a substantial cost in programmer effort.

3.6 Discussion

As previously discussed in Chapter 2, two related problems with existing parallel programming systems are an inappropriate level of abstraction and the explicit nature of communication between processes. On the one hand, high-level abstractions make it easier for the programmer to implement common algorithms and data structures by hiding the low-level details. For example, DSM systems hide the existence of distributed memories by simulating hardware-based shared memory. On the other hand, low-level abstractions allow the programmer to explicitly tune the application for performance by exploiting low-level details of the high-level operation. For example, in message-passing systems the programmer bears the burden and complexity of managing the details of process management and explicit data communication. Since both high-level abstractions and low-level optimizations are desirable, the challenge is in providing an efficient framework to capture the high-level semantics and translate them into efficient invocations of low-level mechanisms.

Aurora addresses this challenge by providing high-level abstractions while using the scoped behaviour technique to integrate the various software layers and improve performance. Although the abstractions are high level, they can be efficiently mapped to and optimized at the lowest levels of the system. Programmers view scoped behaviour as an interface to a pre-packaged set of data-sharing optimizations. The system implementor views scoped behaviour as a framework within which both compile-time and run-time information about the parallel program can be exploited for performance optimizations. Even at the lowest run-time system layer of software, the high-level semantic information inherent in different scoped behaviours can be exploited for greater performance.

For example, the application programmer applies the read cache scoped behaviour because it is known that the data is read-only and reused many times. The class library implements the scoped behaviour by transparently changing the code generated for accessing the data to use a cache and by dynamically loading the cache before it is required. And, the run-time system can switch to a more efficient bulk-data transfer protocol to communicate the data since the scoped behaviour encapsulates the fact that all of the data is to be accessed. Through all the various software layers, scoped behaviour makes it possible to the optimize a parallel program's performance.

3.7 Summary

We have discussed the programmer benefits of Aurora's novel scoped behaviour abstraction. In subsequent chapters, the implementation techniques introduced by the Aurora system and the performance benefits made possible by scoped behaviour will be detailed.

Aurora uses C++ classes and objects to provide a convenient parallel programming environment. Aurora also uses scoped behaviour to integrate the various software layers, from the highest level of abstraction to the lowest level implementation detail, to improve performance. Notably, both data parallelism and task parallelism are supported within the same program. Although Aurora is not the first system to integrate different process models, it is a capability that is often missing in existing parallel programming systems with important consequences in terms of performance (Chapter 5).

Aurora's shared-data ADTs are accessed using the same C++ notation as normal data structures despite being distributed across different memories. In this way, Aurora is similar to traditional DSM systems. The programmer can ignore the details of how shared data is communicated until the program is debugged, and then scoped behaviour can be used to incrementally tune performance. Unlike traditional DSM systems, Aurora can optimize many different data-sharing patterns and different optimizations strategies can be devised by combining scoped behaviours on a per-object and per-context basis. And, as we have seen, scoped behaviours can be applied, removed, and replaced with only mildly intrusive changes to the program. Unlike message-passing systems, different optimization strategies can be investigated with a minimum number of error-prone changes to the source code. This level of flexibility for optimization and experimentation is an important advantage of Aurora over existing systems.

Chapter 4

Scoped Behaviour: Design and Implementation

Now that we have introduced the basic abstractions in Aurora, we can focus on the design and implementation of the shared-data abstract data types (ADT) and scoped behaviour. In the next chapter, we experimentally evaluate the performance of applications written using Aurora.

As previously discussed, the scoped behaviour abstraction transcends all the layers of software in Aurora and, consequently, can be viewed in a number of different ways. To the application programmer, scoped behaviour is how an optimization is applied to a shared-data object. But, to the implementor of Aurora's class libraries, scoped behaviour is a change in the implementation of an ADT's interface methods for the lifetime of a language scope [Lu, 1997b, Lu, 2000]. Different interfaces implement different optimizations.

Of particular interest is how scoped behaviour is implemented in C++ without special compiler support and without language extensions. As a result, scoped behaviour requires less engineering effort to implement and extend than many existing parallel programming systems. Scoped behaviour's implementation framework provides a significant amount of per-object and per-context optimization flexibility within standard C++.

4.1 Handle-Body Composite Objects

Some of the ideas behind scoped behaviour have been explored as part of the handle-body and envelope-letter idioms in object-oriented programming [Coplien, 1992], the strategy design pattern [Gamma *et al.*, 1995], and parametric shared regions in ABC++ [O'Farrell *et al.*, 1996]. Scoped behaviour builds upon these ideas.

Composite objects, such as handle-body objects, are multiple objects that behave as if they were a single entity [Coplien, 1992]. In Aurora, the *handle object* defines the programmer's

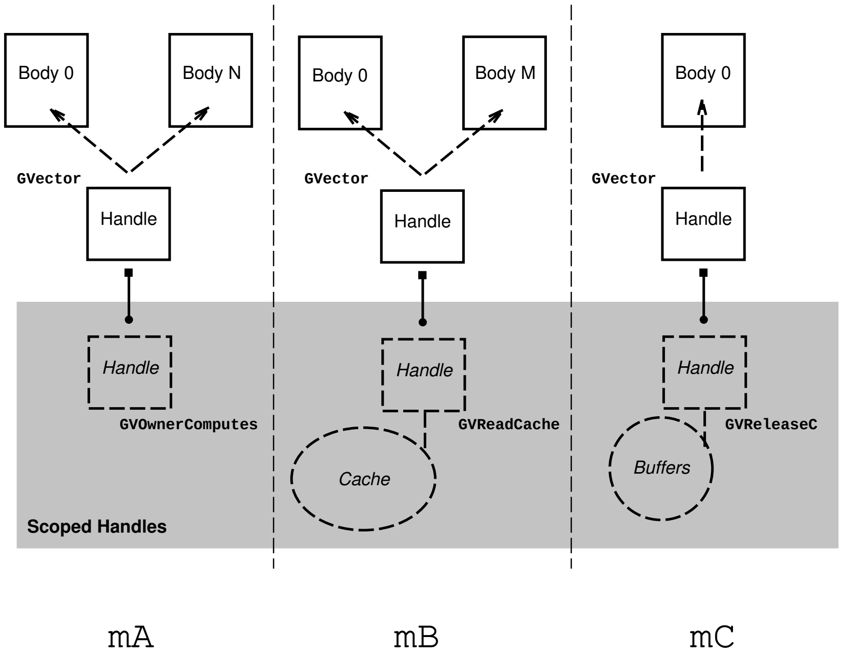


Figure 4.1: Example of Composite Objects in Matrix Multiplication

interface to the shared data and the *body object* (or objects) contain the actual data. A handle object can also be viewed as an object-oriented data descriptor. Having multiple handles for the same body objects is a convenient way to support different ADT interfaces to the same encapsulated data. Depending on which handle is in use, the methods and behaviours will be different. The interaction between handle and body is discussed further in Section 4.4, but we first focus on how different handles interact.

4.2 Language Scopes and Scoped Handles

Language scopes are used to define the context of scoped behaviour in order to exploit the compile-time property of *name hiding* and the run-time properties of object creation. In many block-structured languages, an identifier can be reused within a nested language scope, thus hiding the identifier outside of the scope. A handle within a language scope that hides a handle outside of the scope is called a *scoped handle*.

Figure 4.1 shows the relationship between the various collaborating objects inside the language scope of the matrix multiplication example. The solid boxes show the original `GVector` handle-body objects for `mA`, `mB`, and `mC`. In the example, each of the `GVectors` have a different number of body objects.

The dashed boxes (highlighted in gray) show the scoped handles used to implement the

(a) Scoped Behaviour Macro

```
#define NewBehaviour( ORIG, SB, TYPE ) \                // Macro provided by aurora.H
  GPortal< GVector<TYPE> > AU_ ## ORIG( ORIG ); \
  SB< TYPE > ORIG( AU_ ## ORIG );
template <class C_OrigHandle>                           // Class template provided by aurora.H
class GPortal
{
  private:
    C_OrigHandle * save;                                // Saved handle
  public:
    GPortal( C_OrigHandle & h ) { save = &h; }          // Constructor (input)
    operator C_OrigHandle &() { return *save; }         // Type constructor (output)
}; // GPortal
```

(b) Source Code

```
GVector<int> vector1( 1024 );
{ // Begin new language scope
 NewBehaviour( vector1, GVReleaseC, int );

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

(c) After Standard Preprocessor Pass

```
GVector<int> vector1( 1024 );
{ // Begin new language scope
 GPortal<GVector<int> > AU_vector1( vector1 );
 GVReleaseC<int> vector1( AU_vector1 );
 for( int i = 0; i < 1024; i++ )
    vector1[ i ] = someFunc( i );
} // End scope
vector1[ 0 ] = 1;              // Synchronous update (still)
```

Figure 4.2: Aurora's Scoped Behaviour Macro

owner-computes, caching for reads, and release consistency behaviours. Inside the scope, the `GVector` handles are hidden and unused, but the scoped handles can access the data in the body objects via a reference (e.g., pointer) to the original handles. Dynamic actions can be associated with the construction and destruction of the scoped handles, such as creating, flushing, and destroying cache and buffer objects.

4.3 Handle References and Alternate Interfaces

As shown in Figure 4.2(a), Aurora provides the scoped behaviour macro `NewBehaviour` and the class template `GPortal` to help establish the reference from one handle to another. As discussed below, class template `GPortal` is never used directly by the application programmer; it is used internally by Aurora. Figure 4.2(b) shows the original programmer's source code and Figure 4.2(c) shows the code after the standard preprocessor of the C++ compiler has expanded the macro. Again, the code is shown side-by-side for comparison.

The `NewBehaviour` macro is parameterized by the name of the original shared-data ob-

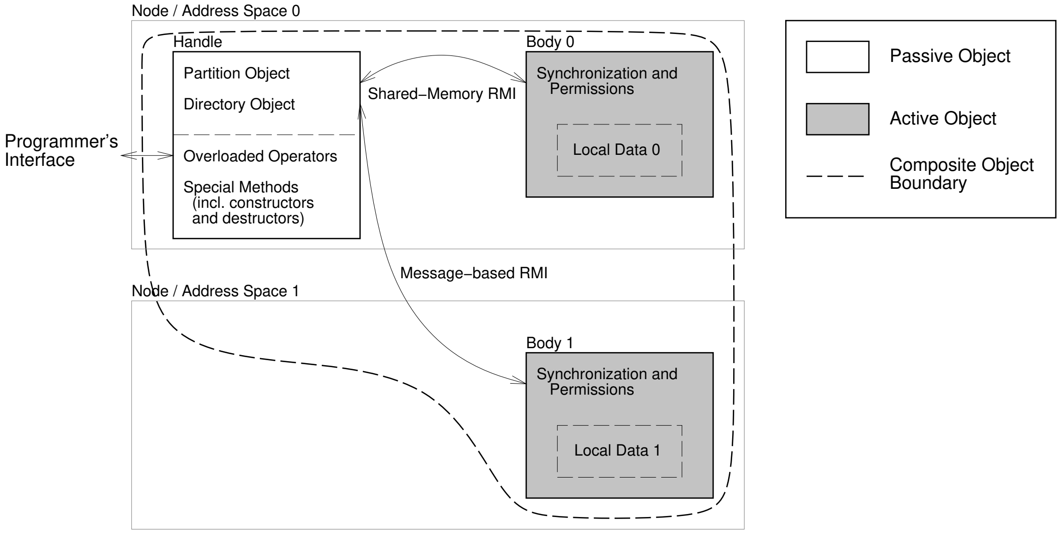


Figure 4.3: Shared-Data Composite Objects

ject (`ORIG`), the scoped behaviour or type of the new scoped handle (`SB`), and the type of the vector elements (`TYPE`).[1] The macro actually instantiates two objects. The first object, `AU_vector1`, is of type `GPortal`. Its sole function is to cache a pointer to the original object. The second object, the scoped handle `vector1` of type `GVReleaseC<int>`, hides the original object but can access its internal state using the pointer passed by `AU_vector1`. Thus, the scoped handle can delegate, mimic, or change the functionality of the original shared-data object and the user's source code does not change. In other words, the programmer's interface of the ADT is changed without changing the encapsulated data or the user's source code.

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

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

[1]Note that it is a multi-line macro and the `##` symbol is the standard preprocessor operator for lexical concatenation. Also, the prefix `AU_` is arbitrary and can be redefined, if necessary. Unfortunately, the more concise syntax of `GVReleaseC<int> vector1( vector1 )` conflicts with the semantics of the C++ standard. According to the standard, the new `vector1` is passed a reference to itself, instead of to the original object, so an intermediate `GPortal` object is required. Fortunately, the macro hides the existence of the intermediary object.

4.4 Shared-Data Composite Objects

In this section, we take a detailed look at the design and implementation of the C++ classes for the shared-data objects and data-sharing optimizations. By design, these classes collaborate to support scoped behaviour.

As discussed, the class library uses the handle-body idiom to create composite objects for shared data (Figure 4.3). In addition to simplifying the implementation of scoped behaviour, the extra level of indirection between handle and body allows for:

1. *Data distribution.* In the simplest case, a distributed scalar has a single body object. However, a distributed vector is a set of body objects, each of which can be located in a different address space or on a different processor node. If there are multiple body objects, the handle includes a *partition object* to abstract the distribution strategy and a *directory object* to keep track of the location of the bodies.

   Given a vector index, the partition object returns the identifier of the body object that stores the vector element. When the distributed vector is created, an appropriate partition object is selected for the desired data distribution. Given a body object identifier, the directory object is used to locate the specific body object, wherever it might be.

   Figure 4.3 shows a distributed vector object with a handle and two body objects, where one of the body objects is on a different processor node than the handle.

2. *Location-transparent data accesses.* Through overloaded operators in the handle, the distributed data can be accessed through a uniform interface, regardless of the location of the actual data. Thus, for a given vector index, the partition object determines which body holds the data and the directory object provides a pointer to the body object.

3. *Cheaper parameter passing of shared data.* Only handles are passed across function calls; the data in the bodies are not copied. Handles can also be passed between address spaces, if desired, since the partition and directory objects are sufficient to locate any body object from any address space.

For performance-sensitive functions, such as `dotProd()` in Figure 3.3, the overheads of handle-body indirection can be avoided in controlled ways through type constructors that return C-style pointers.

The current implementation of Aurora creates handles as passive (i.e., regular) C++ objects. However, each individual body is implemented as an active object, an object with its own thread of control, which is useful for implementing any necessary synchronization behaviour. The

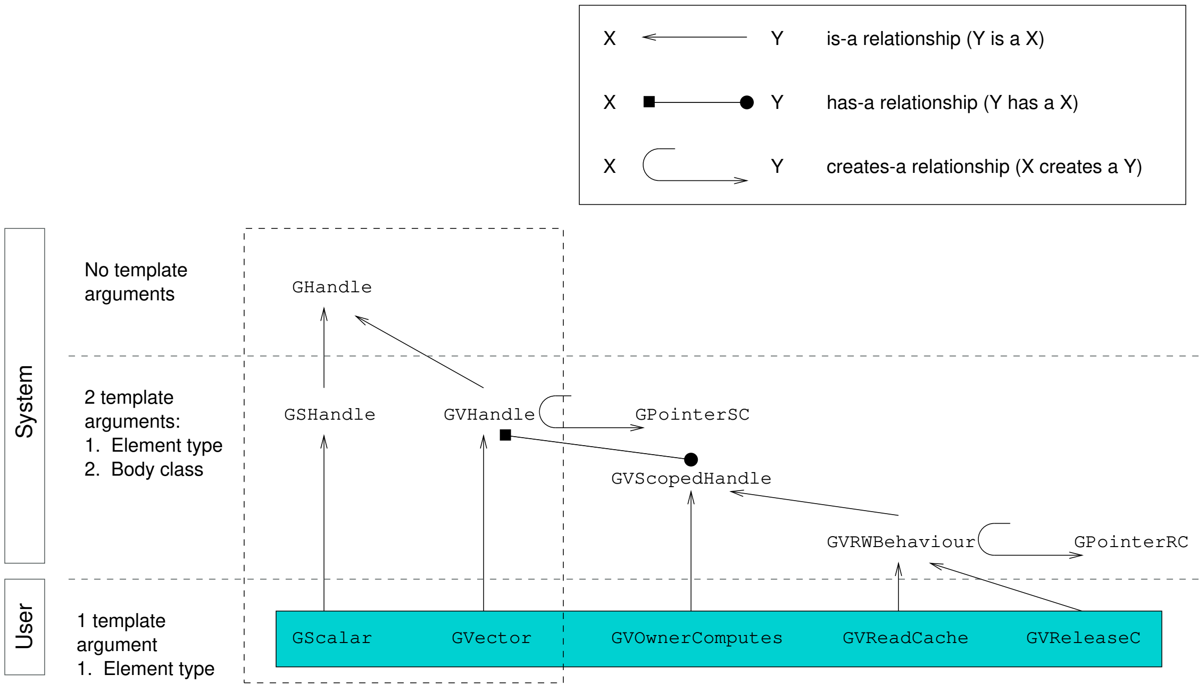


Figure 4.4: Class Hierarchy for Handles

body classes support `get()` and `put()` data access methods, including batch-update and block-read variations. Handle and body interact using the remote method invocation (RMI) mechanism provided by ABC++ [O'Farrell *et al.*, 1996]. The run-time system automatically selects between shared-memory and message-based communication (i.e., MPI) mechanisms for transmitting RMIs.

4.5 Class Hierarchy for Handles

Since most of the data-sharing functionality is implemented in the handles, this discussion will focus on the handle classes. Figure 4.4 and Figure 4.6 are diagrams of the main classes in the hierarchy of shared-data handles.[2] We will focus on Figure 4.4 for now. In general, the application programmer is only expected to use the classes at the leaves of the hierarchy (labelled "User" and highlighted in gray). These classes hide the more complex templating and class hierarchy considerations that the "System" must deal with. The dashed box contains the key classes for the basic shared-data object functionality. From the user's point of view,

[2]The notation is based on that of Grady Booch [Booch, 1991] but with some simplifications and changes to better suit this presentation.

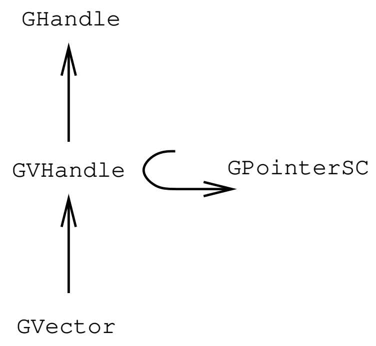


Figure 4.5: Class Hierarchy for `GVector`

instantiating objects of class `GScalar` and class `GVector` is all that is required to use the DSD functionality.

Figure 4.4 shows the specific relationship between classes from the system implementor's point of view. The *is-a* relationship is the usual notion of inheritance. Class Y is a subclass of X so "Y is a X." For example, `GVHandle` (`V` is for vector) is a subclass of `GHandle`, so an object of class `GVHandle` is also of class `GHandle` (see also Figure 4.5). In fact, class `GHandle` is the base class for all handles. Common access methods are factored into the base class.

The *has-a* relationship exists when an object contains a reference or pointer to an instance of another class. If "Y has a X," then an object of class Y contains a reference or pointer to an object of class X. And, assuming the right access control permissions (e.g., `friend` in C++), Y can access the internal state of X using that pointer. In general, the has-a relationship also includes the case where Y contains an instance of X. However, in these classes from Aurora, the has-a relationship is always implemented using a pointer and not an instance.

The has-a relationship enables a programming technique called *delegation*, in which a call to one method is redirected to a second method. In essence, the implementation of first method is, ultimately, a call to the implementation of the second method. Sometimes, the first method contains nothing more than the method call. At other times, there may be some wrapper code before and after the redirected method call. Inheritance is an *implicit* technique to reuse the implementation of a method of a base class in the derived class; delegation is an *explicit* technique to reuse the implementation of a method. With delegation, the user is responsible for writing the code to redirect the method call. In an object-oriented context, this allows a class to have the same interface methods as another class, but the implementation of the methods is not repeated in both classes. Also, a class can selectively redefine or delegate behaviour on a

Framework	`GVector`	**Data-Sharing Optimizations**		
		`GVOwnerComputes`	`GVReadCache`	`GVReleaseC`
Internal State	Partition and directory objects	Handle reference, pointer to local data	Handle reference, cache object	Handle reference, buffer objects
Enter scope (Constructor)	Create body objects	Save reference, find local data	Save reference, create and load cache	Save reference, allow buffer creation on demand
Within scope	Synchronous data access using `GPointerSC`	Access local data using pointer	Read from cache, synchronous updates using `GPointerRC`	Buffer updates, synchronous reads using `GPointerRC`
Exit scope (Destructor)	Free body objects		Free cache	Flush and free buffers

Table 4.1: Object Model for Vector Handles

method-by-method and operator-by-operator basis.

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

The C++ object model provides a convenient create-use-destroy framework within which to implement composite objects (Table 4.1). For the basic shared-data vector, the relevant classes in the hierarchy are `GHandle`, `GVHandle`, and `GVector` (Figure 4.5 and Figure 4.7). When the application programmer instantiates a `GVector`, the class constructor transparently creates the body objects. Later on, the destructor automatically frees the body objects. During the lifetime of the object (i.e., within scope), class `GVHandle` contains the partition and directory objects needed to locate and communicate with the body objects.

To read and write data, the overloaded subscript operator of `GVHandle` returns an object of type `GPointerSC`, which is a *pointer object*. When evaluating C++ expressions involving objects and overloaded operators, temporary objects represent the result of subexpressions [Coplien, 1992]. Since the actual data for a term may be a remote shared data element, the temporary object points to the body object with the data. Class `GPointerSC` has data members to store the vector index and a pointer to the specific body object with that element. Reading from or writing to the vector element invokes the appropriate type constructors and the overloaded assignment operator of `GPointerSC`, resulting in a synchronous

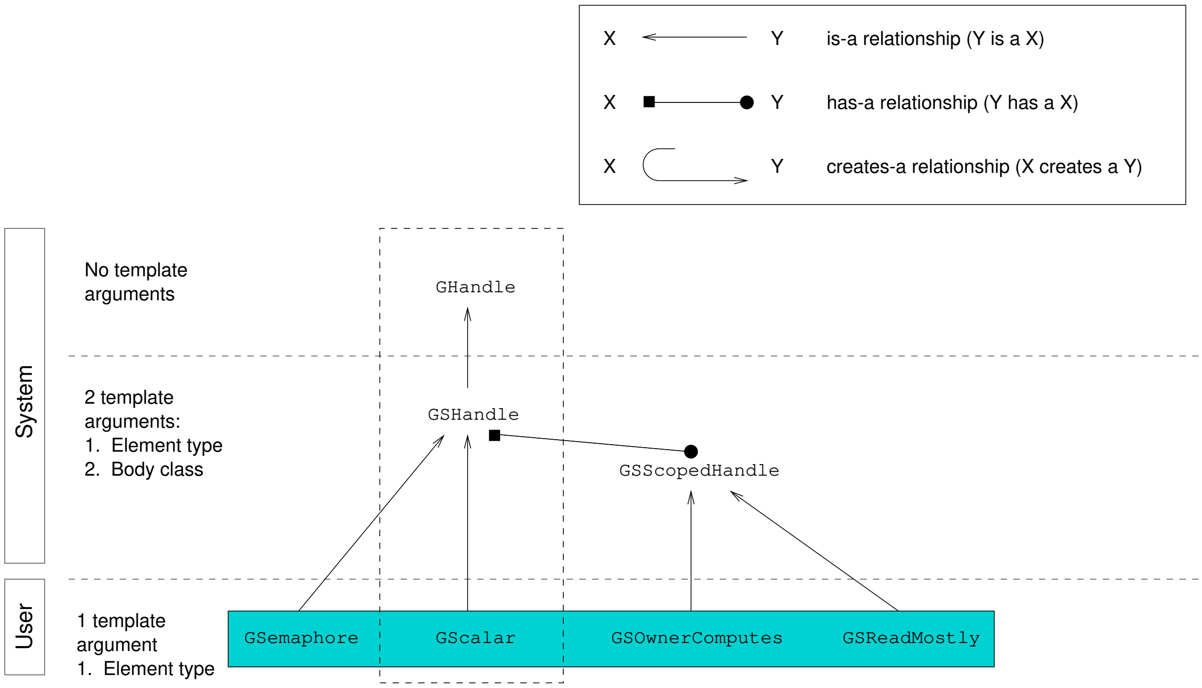


Figure 4.6: Class Hierarchy for Scalar Handles

remote memory access.

So far, we have focussed on the vector classes, but scalar objects also provide important functionality. Of course, there is the basic `GScalar`, which is part of a hierarchy including classes `GHandle` (same as for vectors) and `GSHandle` (Figure 4.4 and Figure 4.6). Orthogonally, class `GSemaphore` is used to create a global semaphore object for synchronization and mutual exclusion. (Figure 4.6 and Table 4.2). As one would expect, different critical sections can be created using different semaphore objects. Although a semaphore is a more specialized shared-data object than `GScalars`, its implementation is similar to shared-data scalars and so it is part of the `GHandle` and `GSHandle` class hierarchy.

4.6 Data-Sharing Optimizations: Scoped Handles

The create-use-destroy object model is combined with handle references to implement scoped handles (Table 4.1 and Table 4.2). For data-sharing optimizations on vectors, the parent class `GVScopedHandle` extracts and maintains a reference to a `GVHandle`, as per the has-a relationship (Figure 4.4 and Figure 4.8). The partition and directory objects of the `GVHandle` are not copied, thus minimizing the construction costs of a scoped handle.

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


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


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

Figure 4.7: Interface for Shared Vector: `GVector`

Framework	GSemaphore	GScalar	Data-Sharing Optimizations	
			GSOwnerComputes	GSReadMostly
Internal State	Single local body, semaphore	Single local body	Handle reference, pointer to local data	Handle reference, cache object
Enter scope (Constructor)	Create local body object and semaphore	Create local body object	Save reference, find local data	Save reference, create, register, and load cache
Within scope	Semaphore operations P(), V(), etc.	Synchronous data access	Access local data using pointer	Read from cache, write updates sent eagerly to all caches
Exit scope (Destructor)	Free body object and semaphore	Free body object		Free and unregister cache

Table 4.2: Object Model for Scalar Handles

The constructor of class GVOwnerComputes uses the handle reference to determine the address of the local (i.e., co-located) body object's data. Therefore, GVOwnerComputes can return a C-style pointer from the appropriate type constructor and from the overloaded subscript operator. The local data is then accessed using the pointer. As previously discussed, GVOwnerComputes also defines special functions to support easy iterating over the local data.

Most of the functionality of the caching for reads and release consistency behaviours are implemented within a cache and buffer manager class. Class GVRWBehaviour can, optionally, create a read cache for shared data and create update buffers to shared data (Figure 4.9). Classes that derive from GVRWBehaviour explicitly configure the caching and buffering options. The overloaded subscript operator in GVRWBehaviour returns an object of class GPointerRC, which is similar in concept to class GPointerSC, but with two important differences. First, if the read cache exists and is loaded, then GPointerRC is configured to access data from the cache instead of from the remote body. Second, if the update buffers are enabled, then GPointerRC is configured to store updates in the buffer rather than initiate a remote memory access. The buffers are created on demand. Depending on the configuration of the cache and buffers, GPointerRC will access shared data appropriately.

Therefore, the constructor of class GVReadCache calls the appropriate GVRWBehaviour methods to create and load the read cache. Similarly, the constructor of class GVReleaseC calls the appropriate GVRWBehaviour method to enable the use of update buffers (Figure 4.9). The destructor for class GVRWBehaviour makes sure all buffers are flushed.

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

Figure 4.8: Handle References: `GVScopedHandle`

For data-sharing optimizations on scalars, the parent class `GSScopedHandle` (which is different from `GVScopedHandle` with a `V`) extracts and maintains a reference to a `GSHandle`, as per the has-a relationship (Figure 4.6). Class `GSOwnerComputes` is similar to its vector counterpart. In keeping with the crease-use-destroy object model (Table 4.2), the constructor of class `GSOwnerComputes` locates the local body object of the scalar data, then access methods in the class can return a C-style pointer to that data. The main difference between vector owners-computes and scalar owner-computes is that there is no need (and no support for) iterating over scalar data.

A second scoped behaviour for scalar objects is implemented by class `GSReadMostly`.[3] A number of researchers have noted the read-mostly, write-sometimes access pattern and the monotonic property of some shared data structures (e.g., [Bennett *et al.*, 1990b, Bennett *et al.*, 1990a, Bal *et al.*, 1992, Kalé and Krishnan, 1996]). For example, in branch-and-bound algorithms, the current best bound is read often to find the cut-offs, but it is updated infrequently when a better bound is found. The Travelling Salesperson Problem (TSP) discussed in Chapter 5 is a branch-and-bound algorithm and exhibits this access pattern. Therefore, we developed a new scoped behaviour that allows a scalar value to be cached for reads and for all cached copies of the value to be eagerly updated whenever a noteworthy new value is written to the shared scalar.

[3] As a special case, we also allow this behaviour to be selected via a method call (i.e., `setEagerUpdate()`). Therefore, when the scalar is a global variable, we can select this new behaviour in the global scope as well.

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

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

Figure 4.9: Interface for Release Consistency Scoped Behaviour: `GVReleaseC`

In contrast to the previous read cache scoped behaviour for vectors, updates to the value are visible in all caches before the end of the scope. The timely arrival of a new value is the motivation for the new behaviour. When a process applies the scoped behaviour to the `GScalar`, a local cache is created on the node of the process and the body object at the shared scalar's home node registers the existence of the new cache and sends the current value. Subsequently, when any process writes to the shared scalar, the new value is sent to the body at the home node. Upon receiving the updated value, the body at the home node eagerly forwards the new value to all the local caches that have been registered. Since body objects are active objects, the propagation of updated values is concurrent and asynchronous with the computation. When the scope is exited, the local cache is destroyed and the body at the home node is told to remove that local cache from its internal list.

It is also possible to select and enforce the criteria under which the updates are forwarded to the local caches. For example, the monotonic decreasing property is selected for the shared best bound in TSP. New bound values sent to the body object at the home node are compared to the current value before any updates are propagated. The upper bound for tours can only decrease, therefore only lower values are of interest. For TSP, if and only if the new value is strictly less than the current value does the body at the home node send the update messages to all the local caches. Other applications may selectively enforce different monotonic properties (e.g., monotonic increasing, monotonic non-decreasing). It is possible for an incoming new value and a previous outgoing update to "pass each other" somewhere in the communications channel (i.e., race condition), but since the monotonic property is guaranteed at the body object, there is no need to create a critical section for updating the bounds value. Without the extra property-checking functionality provided by class `GSReadMostly`, a more traditional programming system would require a lock around the update to guarantee mutual exclusion. But, since the local body at the home node is a serialization point for the new values, only the relevant updates are propagated.

4.7 Extending the Library

Within the class hierarchy, new data-sharing optimizations can be implemented. As a trivial but illustrative example, a new class could cache data for reading *and* buffer updates (Table 3.3). So far, we have only considered the scoped behaviour for reads or writes, but not for both types of accesses. Since improving the locality for both reads and writes can be useful, we will create a new scoped behaviour while reusing as much as the existing classes and infrastructure as possible.

The new class would derive from `GVRWBehaviour` (Figure 4.9). The new class's constructor creates the read cache (i.e., calls `createCache()`) and also enables the update buffers (i.e., calls `allowUpdateBuf()`). The `GPointerRC` objects created by the new class would always read from the cache and always buffer updates. By default, updates are also mirrored in the cache. Admittedly, this "new" data-sharing optimization is easy to add because of the design and existing functionality of `GVRWBehaviour` and `GPointerRC`, but the basic techniques can be used for more complex additions to the library.

There are three main techniques for extending the library of data-sharing optimizations:

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

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

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

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

3. *Redefine methods.* Inherit from a parent class, then redefine behaviour through constructors, the destructor, methods, operators, and type constructors.

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

The techniques can also be combined to create several different new scoped behaviours within the same implementation framework.

4.8 Summary

Previously, we discussed the abstraction benefits of scoped behaviour for the application programmer. The high level of abstraction of Aurora's shared-data objects is not necessarily a liability since scoped behaviour can optimize the performance-critical data-sharing patterns of the ADTs. By combining a set of pre-packaged scoped behaviours with the various shared-data objects of a parallel program on a per-object and per-context basis, a large number of optimization strategies can be supported. But the question remained, how much engineering effort is required to provide such a high degree of flexibility and optimization? Some previous parallel programming systems required pre-processors (e.g., Enterprise), new compilers (e.g., HPF), and special operating system support (e.g., Munin) in order to provide even a basic level of functionality.

In this chapter, we detailed how scoped behaviour is implemented without special compiler support and without language extensions. Scoped behaviour can be implemented using only standard language mechanisms and a class hierarchy. The novel implementation framework developed for scoped behaviour allows the high degree of optimization flexibility to be provided entirely within standard C++. By extending the previous work on handle-body composite objects to include language scopes and scoped handles, scoped behaviour provides an new way to implement multiple ADT interfaces for the same objects. Different interfaces implement different data-sharing optimizations and the semantic information of the scoped behaviours can be carried across the layers of software for even greater performance benefits. The quantification and evaluation of scoped behaviour's performance is the next topic of discussion.

Chapter 5

Performance Evaluation

We now compare and contrast the performance of the same four applications implemented using three different types of parallel programming systems: Aurora (a DSD system with scoped behaviour), TreadMarks (a page-based DSM system), and a message-passing system (either MPICH or PVM, depending on the application). The message-passing version of three of the four applications are implemented using MPICH and originate from the University of Toronto. One message-passing program has been implemented using PVM by researchers from Rice University. Each application exhibits different data-sharing patterns and stresses different aspects of the programming systems. 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.

As we will see, Aurora closely matches, and sometimes significantly surpasses, the performance of message passing. Furthermore, Aurora generally surpasses, sometimes by a wide margin, TreadMarks in performance. In distributed computing, it is generally acknowledged that message-passing programs set a high standard of performance. Therefore, our primary goal is to show that Aurora can approach the level of message-passing performance while still maintaining the usability benefits of a high-level shared-data abstraction. Also, since TreadMarks is a state-of-the-art DSM system, it provides another contemporary performance-oriented benchmark on the hardware platform used in this study.

In our terminology, an *application* is a computational problem as solved by a specific algorithm. For example, parallel sorting is an problem that can be solved by the Parallel Sorting by Regular Sampling algorithm [Li *et al.*, 1993]. A *program* is an implementation of an application using a parallel programming system. For example, the same parallel sorting application can be implemented using different parallel programming systems, and each program exhibits differences according to the models and mechanisms inherent in the programming system. Finally, a *dataset* is a specific set of input data or initial state for an application or program. The same dataset can be used for different programs. By varying the dataset, the quantity and quality of computation performed by a program can change. For example, the parallel sorting application can execute with a dataset of either 6 or 8 million keys, where the amount of computation increases with the number of keys to be sorted. Even for lists of the same size, variations in the key values and ordering of keys in the list can affect the amount and nature of the computation.[1]

There are three recurring themes in this comparison:

1. *The relative performance of the programs using the three systems.* We consistently use the performance of message passing as the baseline for performance. By comparing the three systems on a common hardware platform, we gain insight into the relative strengths and weaknesses of the software systems.

2. *Processor scalability* or *the scalability of a program as the number of processors is increased.* As the degree of parallelism is increased (i.e., more parallel processors are used), the absolute speedup should increase for a given program.

   Ideally, a parallel program should scale (unit) linearly. That is to say, for each additional processor, the speedup should also increase by one. Unfortunately, the speedup of most programs tends to scale at a rate that is less than unit linear due to limitations in the algorithm or in the parallel programming system. As we will see, if one program exhibits poor processor scalability, but the other programs for the same application exhibit good processor scalability, the problem is likely the programming system. Also, a program can experience a reduction in speedup (i.e., a slowdown) as the number of processors is increased. In these cases, a serious bottleneck in the programming systems is often the cause of the slowdown.

3. *Problem scalability* or *the scalability of a program as the problem size is increased.* As the problem size increases, so should the speedup for a the same number of processors.

   Generally speaking, and ignoring the cases where there is paging to disk, as the problem size grows, so does the granularity of computation, which should lead to higher absolute speedups. If a larger dataset does not result in higher speedups, there may be a performance bottleneck in the parallel program.

[1]For example, quicksort is an $O(n^2)$ algorithm if the initial list of n keys is already sorted, and a $O(n \log n)$ otherwise.

5.1 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 3.1), uses the MPICH (version 1.1.10) implementation of the Message-Passing Interface (MPI) as the lowest user-level software layer for communication [Doss *et al.*, 1993]. 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.

Three of 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. One of the message-passing programs, namely the Travelling Salesperson Problem, is implemented using PVM and originates from Rice University. The program uses PVM (version 3.4.1) [Geist *et al.*, 1994] and the program configures the message-passing system to use TCP/IP (i.e., `pvm_setopt(PvmRoute, PvmRouteDirect)`).

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

Application	Dataset	Time (seconds)	Comments
Matrix Multiplication (MM2)	512×512 matrices	110.6 (1 PE) 11.7 (16 PE, MPICH) Speedup of 9.49	Original implementation was for Aurora. Computes $P \leftarrow Q \times R$ then $R \leftarrow Q \times P$. Initial values are randomly generated integers. Allgather data-sharing pattern (Figure 4.1 of [Snir *et al.*, 1996]).
	704×704 matrices	250.7 (1 PE) 25.2 (16 PE, MPICH) Speedup of 9.94	
2-D Diffusion (2DD)	1536×1536, 32 time-steps	31.8 (1 PE) 2.31 (16 PE, MPICH) Speedup of 13.8	Original implementation was for Aurora and based on pseudo-code from POOMA [Reynders *et al.*, 1996]. Initial column of mass at center of grid. Uses `GVector<float>`. Data sharing with two neighbours only.
	2048×2048, 32 time-steps	61.4 (1 PE) 4.50 (16 PE, MPICH) Speedup of 13.6	
Parallel Sorting by Regular Sampling (PSRS)	6 million keys	14.4 (1 PE) 3.68 (16 PE, MPICH) Speedup of 3.91	Original implementation for shared memory [Li *et al.*, 1993]. Keys are randomly generated 32-bit integers. Multi-phase algorithm. Broadcast, gather, and alltoall patterns (Figure 4.1 of [Snir *et al.*, 1996]).
	8 million keys	19.9 (1 PE) 11.21 (16 PE, MPICH) Speedup of 1.78	
Travelling Salesperson Problem (TSP)	"17.large" (17 cities)	12.4 (1 PE) 8,416,840 nodes 1.29 (16 PE, PVM) Speedup of 9.58 10,655,364 nodes 323 tours	Original implementation and all datasets from the TreadMarks group [Lu *et al.*, 1997]. Includes `GVectors` of user-defined datatypes (e.g., `GVector<PrioQElement>` and `GVector<TourElement>`). Master-worker data sharing is most efficient. Performance is highly dependent on the dataset.
	"18b" (18 cities)	15.4 (1 PE) 9,965,463 nodes 1.60 (16 PE, PVM) Speedup of 9.59 12,355,994 nodes 381 tours	
	"20" (20 cities)	38.2 (1 PE) 22,222,235 nodes 6.99 (16 PE, PVM) Speedup of 5.46 23,052,705 nodes 4,154 tours	
	"19b" (19 cities)	51.5 (1 PE) 32,904,280 nodes 4.01 (16 PE, PVM) Speedup of 12.9 34,574,731 nodes 618 tours	

Table 5.1: Summary of Applications and Datasets in Test Suite

No.	Characteristic	MM2	2DD	PSRS	TSP
1	Communication Intensive	Yes	No	Yes	No
2	Main data-sharing pattern(s) (Some terminology based on Figure 4.1 of [Snir *et al.*, 1996])	Allgather	Neighbour	Broadcast, Gather, Alltoall (vector variant)	Master-worker, eager update of scalar
3	Uses explicit data placement	No	No	Yes	Yes
4a	Number of computational phases in algorithm	2	2	4	1
4b	Output of one phase used as input of another	Yes	Yes	Yes	n/a
5	Load balancing is static	Yes	Yes	No	No
6	Data sharing depends on input dataset	No	No	Yes	Yes
7	Original implementation pre-dates this study	No	No	Yes	Yes
8a	Good data-access locality using data parallelism (i.e., SPMD)	Yes	Yes	Yes	No
8b	Good data-access locality using task parallelism (i.e., master-worker)	No	No	No	Yes
9a	Uses shared data of a built-in datatype (i.e., `int` or `float`)	Yes	Yes	Yes	Yes
9b	Uses shared data of a user-defined datatype (e.g., `PrioQElement` and `TourElement`)	No	No	No	Yes

Table 5.2: Characteristic Space of the Applications

5.2 Applications, Datasets, and Justification

As summarized in Table 5.1, the applications are a matrix multiplication program (MM2), a 2-D diffusion simulation (2DD), a parallel sort via the Parallel Sorting by Regular Sampling (PSRS) algorithm [Li *et al.*, 1993], and the travelling salesperson (TSP) problem [Lu *et al.*, 1995, Lu *et al.*, 1997]. These applications have been chosen to demonstrate the expressive power of a programming system and to highlight interesting performance-oriented differences between the systems.

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. The 2-D diffusion application is motivated by the work of computational scientists [Reynders *et al.*, 1996]. Finally, both parallel sorting and combinatorial optimization (e.g., TSP) have been widely studied by researchers because they have 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 5.1.

By design, large portions of source code are identical in the TreadMarks, Aurora, and message-passing implementations of the same application (see Appendices B, C, and D). In this way, differences in performance can be more directly attributed to differences in the programming systems. Note that the original source code for the PSRS and the TSP applications pre-date the development of Aurora. In fact, the TSP application was developed by an independent research group[2] and we have adopted their source code for the TreadMarks and message-passing versions of TSP.

The datasets for matrix multiplication and PSRS are randomly generated. The initial state of the 2-D diffusion simulation consists of a column of mass at the center of the simulated grid. Finally, the datasets for TSP are taken from the TreadMarks research group without any changes.

Four applications are not enough to cover a comprehensive part of the design space of parallel computing. However, the selected applications do cover a variety of interesting data-sharing patterns and characteristics of parallel applications (Table 5.2). Many of the application-specific details are discussed below, but we briefly highlight some of the important characteristics here. The individual applications have been selected to improve the diversity of the experimentation (e.g., Characteristics 2, 3, 4a, 4b, 8a, and 8b), to demonstrate the generality of the Aurora system (e.g., Characteristics 8a, 8b, 9a, and 9b), and to ameliorate Aurora-specific

[2] Co-author "H. Lu" on the TreadMarks papers is not related to this author.

biases in the selection of applications (e.g., Characteristic 7).

For example, each of the four applications contains different data-sharing patterns (Characteristic 2), such as broadcast, neighbour, master-worker, and various examples of collective communication [Snir *et al.*, 1996]. In fact, PSRS contains a number of different data-sharing patterns within the same application and is a noteworthy experimental data point for that reason. Also, both data parallel and task parallel applications are included (Characteristic 8a and 8b). There are, of course, many more data-sharing patterns and forms of parallelism of interest to the community that are not represented by the selected applications. Although we discuss how scoped behaviour and Aurora can be applied to other applications in Chapter 6, much of that remains as future work.

As well, in designing a test suite of applications, there is the risk of biasing the choice of application and the implementations to favour a given system. Although these biases cannot always be eliminated, they can be ameliorated. Towards that, PSRS and TSP have been selected because both their algorithms and original implementations (Characteristic 7) existed before the start of this current study. The original PSRS source code is for the shared-memory BBN TC2000 multiprocessor and was ported to Aurora. In contrast, the matrix multiplication and 2-D diffusion applications were originally developed for Aurora and then ported to the other systems. And, as previously discussed, the TSP source code and datasets are taken directly from an external research group. Overall, the four applications are a modest but substantial attempt to address the diversity of the design space, the generality of the Aurora system, and the potential biases of an experimental study.

5.3 Methodology

The speedups of all the programs are computed against sequential C implementations of the same algorithm (Table 5.1). 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.

Unless noted otherwise, the reported real times are the averages of five runs executed within a single parallel job.[3] 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 error, the activity of other processes on the system (e.g., daemon processes, disk I/O), and other factors (to be discussed later) can cause small variations in the

[3] A parallel job is the execution of a program from the Unix command line. Within a parallel job, we loop and execute the same core code five times.

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 are low relative to the total run time.

To illustrate the process of averaging over five runs, let us consider the case of the matrix multiplication application, with four processors, and the 704 $\times$ 704 dataset.[4] The data points correspond to the four processor results in Figure 5.2 and are based on a sequential time of 250.7 seconds (Table 5.1). All the numbers below are given to two decimal points for consistency. The TreadMarks program recorded real times (all in seconds) of 69.27, 69.92, 69.70, 69.82, and 69.90 over five consecutive runs for an average of 69.72 seconds and a speedup of 3.60. The Aurora program recorded real times of 65.83, 65.73, 65.81, 65.84, and 65.73 over five consecutive runs for an average of 65.79 seconds and a speedup of 3.81. The MPICH program recorded real times of 68.87, 68.89, 68.71, 68.71, and 68.78 over five consecutive runs for an average of 68.79 seconds and a speedup of 3.64. Although the range of real times over five runs can be as high as 0.63 seconds (i.e., TreadMarks), it is still a relatively small range compared to the total run time.

Variations have also been be observed between the average time of one parallel job and another parallel job using the exact same executable and dataset. These variations are believed to be due to page placement and data cache conflicts [Bugnion *et al.*, 1996]. Since there is no (known) way to control page placement under AIX, parallel jobs are repeated multiple times and the best time (which is still an average of five runs) is reported. Although the times reported are "best cases," it should also be noted that the reported real times are also readily reproducible.

Again, let us consider the case of the matrix multiplication application, with four processors, and the 704 $\times$ 704 dataset. In the same benchmarking session that produced the reported result of 65.79 seconds for Aurora, other jobs using the same Aurora executable produced times of 67.95 seconds (average of 67.88, 67.99, 67.92, 68.00, and 67.97) and 68.17 seconds (average of 68.05, 68.06, 68.06, 68.07, and 68.60). Similarly, another job using the same MPICH executable produced a time of 69.87 (average of 70.00, 69.98, 69.78, 69.85, and 69.73).[5] The consistency of the numbers *within* a job, but the variation between jobs suggests that some aspect of resource allocation at the start of a job can have a significant influence on the run times. As discussed above, one possible explanation might be the page placement and data cache conflicts that result from how memory is allocated by the operating system when a job

[4] This illustrative data point is used because all of the measurements are from the same day (July 2, 1999).

[5] The automated benchmarking was truncated due to a hardware fault (`kerpow.eecg.utoronto.ca` crashed), therefore the timing of an alternate job for the TreadMarks program from July 2, 1999, is not available.

is first initialized.

Finally, if the data values of a dataset are randomly generated (as noted in Table 5.1), 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.

5.4 Overview

Throughout this discussion, the message passing 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 message-passing programs are not always the fastest, as can be seen in Table 5.1, they define the baseline in order to be consistent.

Overall, the performance of Aurora programs are comparable to message-passing programs, which is encouraging for Aurora. In some cases, message passing is between 1% to 27% faster than Aurora. In other cases, the performance of Aurora exceeds message passing by margins of between 8.1% to 52% and more in some pathological cases. We define pathological cases to be when there is a significant degradation in performance that is *likely* due to an implementation bottleneck instead of a fundamental flaw with a given programming model. Whenever possible, our discussion will try to distinguish between fundamental flaws and pathology.

We will discuss the reasons behind the performance differences. Of course, Aurora's programming model is significantly different than that of message passing. So, despite a higher-level programming model, Aurora is able to achieve a comparable level of performance to message passing. It is important to note that although the message-passing programs have been tuned for performance (discussed below), further improvements may still be possible.

When compared with TreadMarks programs, the performance of Aurora is usually higher, and sometimes by a significant margin (e.g., 25% to 340%). In some pathological cases, Aurora is faster than TreadMarks by up to a factor of 14. In contrast, out of 40 different performance data points, there are only 2 data points for which TreadMarks beats Aurora in performance, and then by a margin of no more than 8.5%. Our experiments indicate that there are some scalability problems with TreadMarks for 8 processors or more on our hardware platform.

5.4.1 Matrix Multiplication

Design and Implementation

The matrix multiplication application used for this evaluation is different from the program discussed in Chapter 3 (Section 3.4) 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. 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 is 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 is appropriately distributed, as outlined in Section 3.5. 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 (Section B.1). 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 Figure 3.3, 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" (Figure 4.1 of [Snir *et al.*, 1996]). 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)" [Snir *et al.*, 1996].

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 non-blocking sends and receives (e.g., `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 Figure 5.1 and Figure 5.2. 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 a 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 ex-

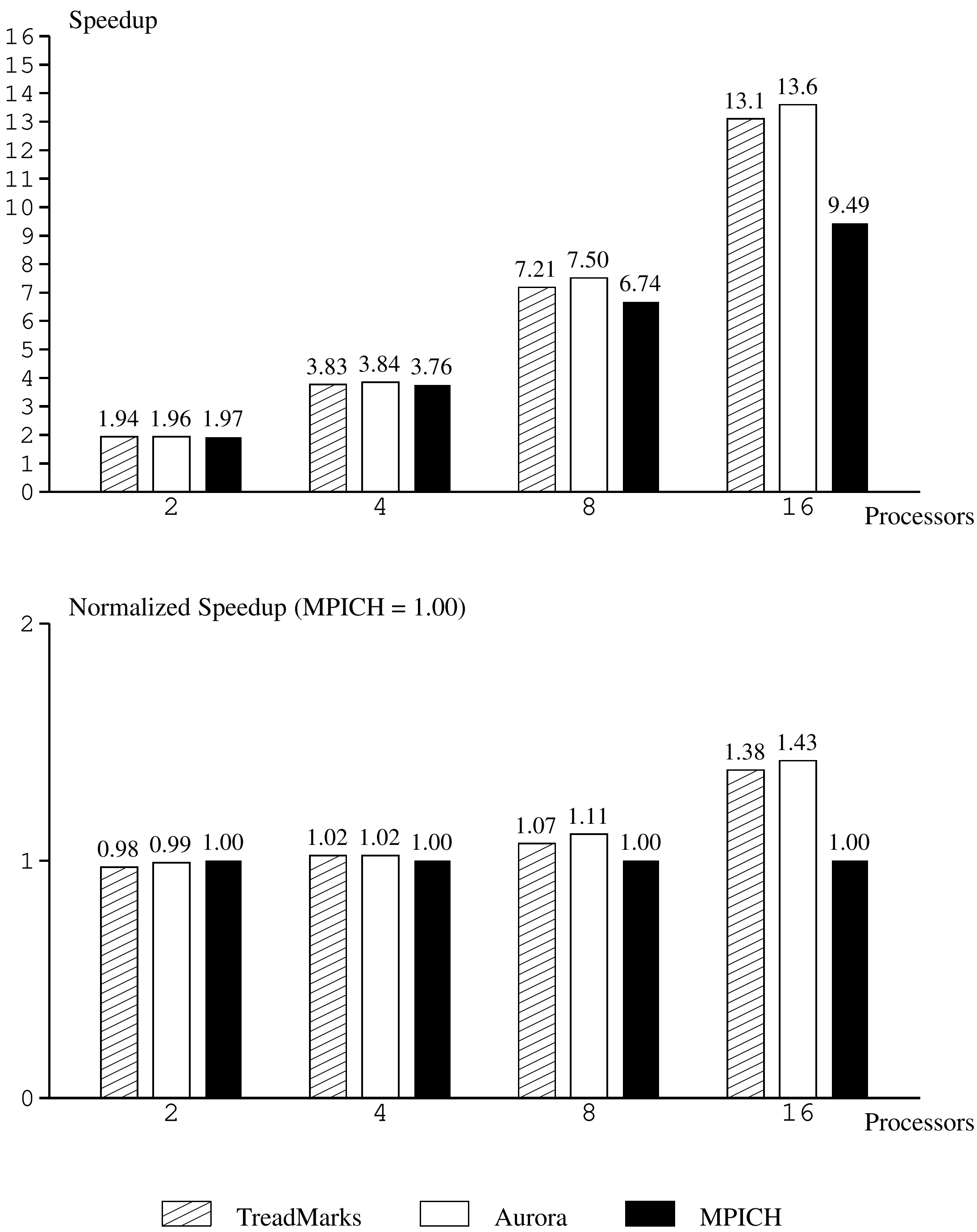


Figure 5.1: Speedups for Matrix Multiplication, 512 × 512

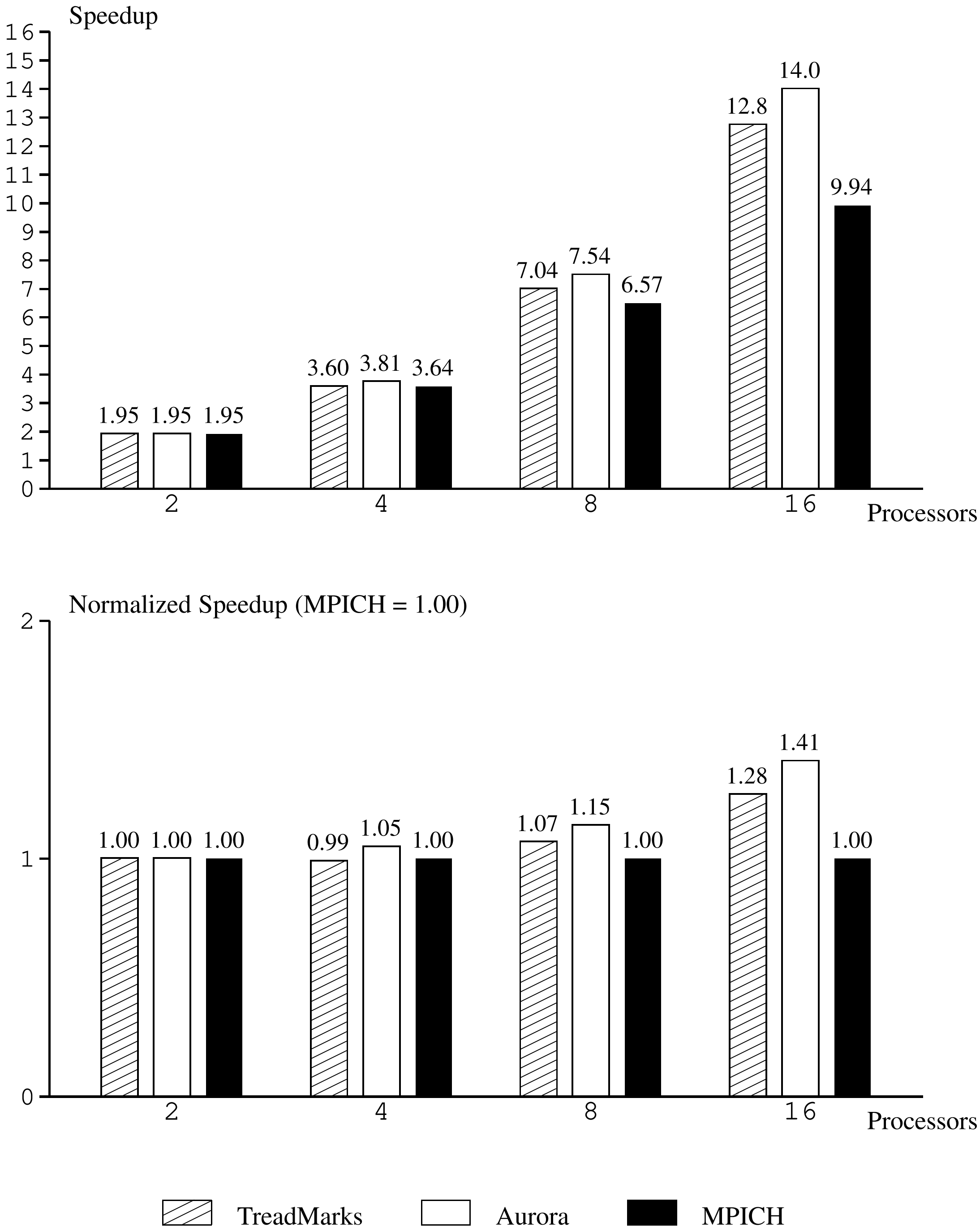


Figure 5.2: Speedups for Matrix Multiplication, 704×704

tent, 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 $\times$ 704 dataset requires 250.7 seconds of sequential execution time, it is still a relatively small computational problem. For example, perfect unit linear speedup on 16 processors requires a run time of 15.7 seconds for the 704 $\times$ 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 (Chapter 2). 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 [Parsons *et al.*, 1997] are more expensive than the simpler approach taken in Aurora. There is no consistency model *per se* in Aurora; rather, data is transferred, manipulated, and made consistent on a language scope basis according to the create-use-destroy model of shared-data objects (Chapter 4).

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 $\times$ 704 dataset than for the 512 $\times$ 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. In the forthcoming discussion of the TSP application, the impact of the protocol overheads will be even greater than for matrix multiplication.

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 $\times$ 704 dataset are shown in Figure 5.3 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[6] 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

[6]A 704 $\times$ 704 matrix of 32-bit integers requires 1,982,464 bytes (or 1.89 MB) of storage. Only a $\frac{1}{16}$ block of the matrix is local to each processor. So, $\frac{15}{16}$ of the matrix (i.e., 1.77 MB) must be read in from remote nodes and each of the 15 remote nodes wants a copy of each node's local $\frac{1}{16}$ block.

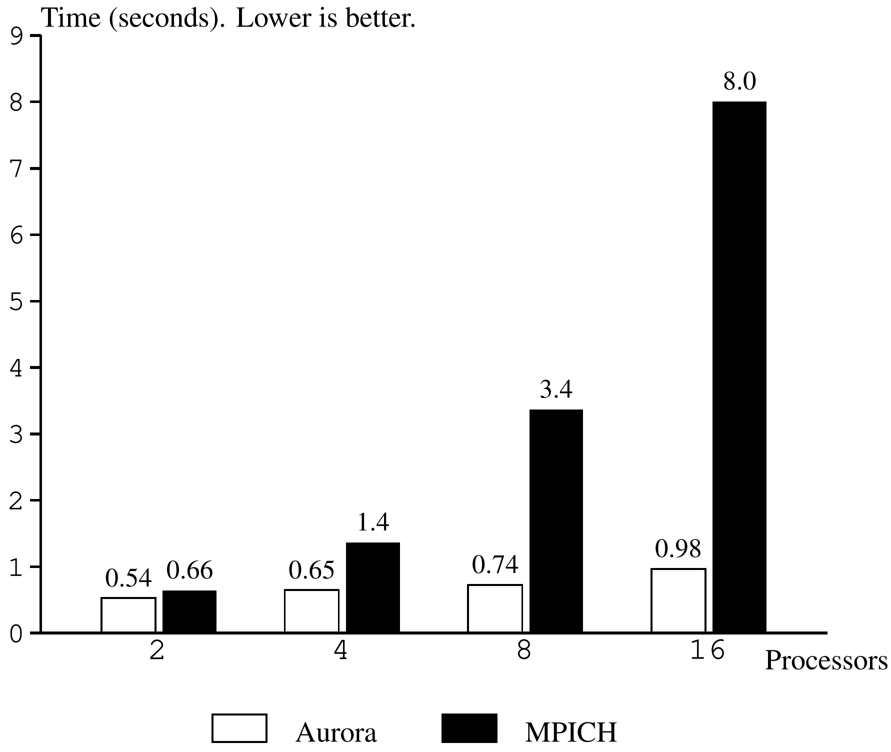


Figure 5.3: Data-Sharing Overheads in Matrix Multiplication, 704 × 704 , Aurora vs. MPICH

---

barriers were added to the Aurora program: one barrier just before 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 5.3 shows that the real-time overheads of Aurora are between 12% (0.98 seconds versus 8.0 seconds) and 82% (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 [Balakrishnan *et al.*, 1998]. 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 acknowledgements, windowed flow control, and slow start, can be complex, especially for our data-sharing pattern [Stevens, 1995, Wright and Stevens, 1995]. 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 suggests 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 a pathological 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.

5.4.2 2-D Diffusion

Design and Implementation

Simulating the diffusion of matter involves data-sharing idioms that are archetypical for a different class of applications than matrix multiplication. Therefore, it is important that scoped behaviour be able to support idioms beyond the standard read-intensive and write-intensive patterns. The 2-D diffusion application also demonstrates the generality of Aurora's class templates by using shared-data vectors of floating point elements (i.e., `GVector<float>`) instead of integers. The same class templates are used for both integers and floating point numbers.

In our simplified simulation of 2-D diffusion, each floating point element of a matrix represents the density of matter at a grid coordinate [Reynders *et al.*, 1996]. Logically, two matrices, `timeNow` and `timePrevious`, are used. For each time-step, the new density of matter at

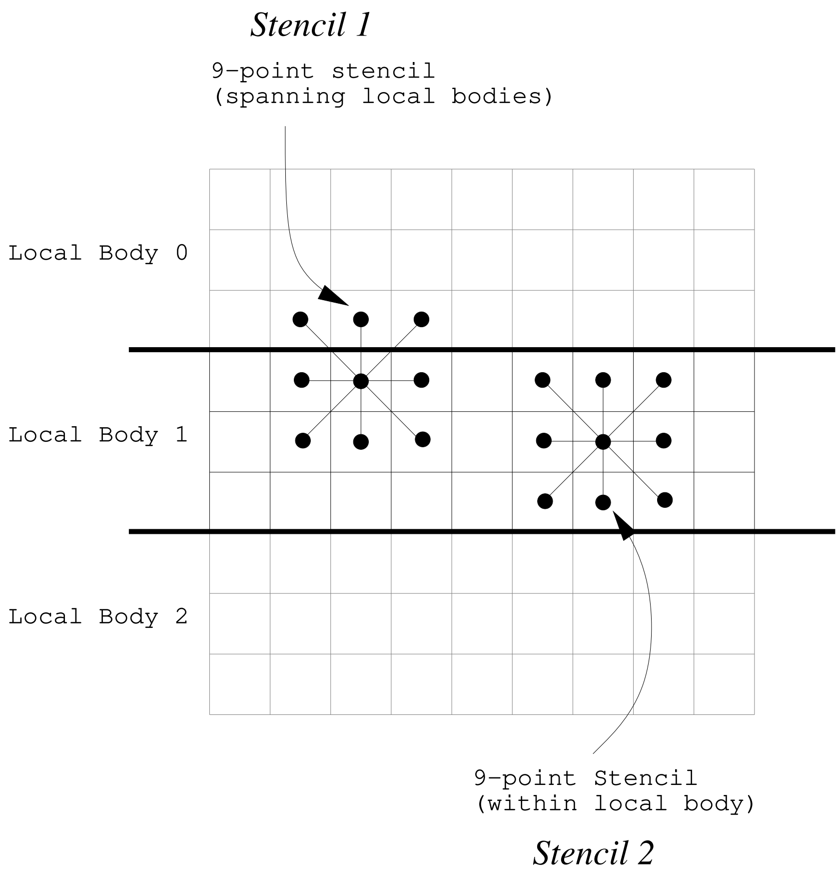


Figure 5.4: Stencils in 2-D Diffusion Simulation

a coordinate in `timeNow` is computed by averaging the densities in `timePrevious` at the same coordinate and at its eight nearest neighbours. This computation is known as a 9-point stencil operation (Figure 5.4). For the next time-step, the two matrices are logically swapped. An important characteristic of the stencil operation is that nearby points in a mesh or grid structure are evaluated in a specific pattern. And, given a reasonable problem size, the nine data points usually lie within the same local body (Figure 5.4, Stencil 2).

Assuming a block-distributed matrix, the normal form of Aurora's 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 neighbours above or below the current coordinate can reside in a different local body (Figure 5.4, Stencil 1). Aurora addresses this form of data sharing by supporting block-distributed matrices with border padding and a modified form of owner-computes. In the literature, the border padding is sometimes referred to as ghost data. The padding extends the local body by one row above and one row below the normal local body. In this way, Stencil 1 can be computed by accessing the top padding and the relevant points in Local Body 1; Stencil 1 does not need to access Local Body 0 directly. Otherwise, these matrices are identical to normal block-distributed matrices. Furthermore, as part of the scoped behaviour, the system makes it easy to load the boundary value data at the borders from the appropriate local bodies (i.e., `fetchPadding()`).

In essence, the new scoped behaviour establishes a producer-consumer relationship between two neighbouring local bodies. All local bodies are producers. The consumers are the neighbouring local bodies. The data that is produced is moved into the border padding storage of the consumer as part of the scoped behaviour's functionality. Once the data movement is complete, the computationally-intensive 9-point stencil operation can be efficiently performed using C-style pointers on the local body plus padding.

Given Aurora's multithreaded run-time system, the current scoped behaviour uses asynchronous and concurrent data movement to transfer the border data. The consumer grabs the data without the explicit participation of the producer process (i.e., it is asynchronous with respect to the producer) and, through the use of threads, data can be sent and received concurrently. An advantage of the asynchronous approach, sometimes referred to as one-sided communications [Message-Passing Interface Forum, 1997], is that it supports both SPMD and non-SPMD execution. When using a non-SPMD process model, it is not always convenient or possible for the producer of the data to synchronize and explicitly send the data to the consumer. An advantage of concurrency is that, if a thread is blocked waiting for data, another thread can still send data. One drawback of the asynchronous approach is that a barrier is required at the end of a time-step to prevent a processor from proceeding too early to the next time-step.

Of course, the padding and ghost data strategy used by Aurora is not new. However, Aurora abstracts the implementation details of the padding and simplifies the data movement using scoped behaviour. Also, in keeping with the abstract data type approach, the new scoped behaviour is general-purpose and can be reused in other programs and with different process models.

The TreadMarks version of 2-D diffusion is very similar to the Aurora version. Although there is no explicit owners-computes in TreadMarks, the loop parameters of each process implicitly partitions the data into disjoint blocks and the processes write to the shared pages of the grid with a high degree of locality. By logically assigning the rows of the grid to processes in a block-distributed fashion, the data will automatically migrate and become block distributed to each process after the first time-step. The only source of data sharing occurs on the borders of each process's portion of the simulated grid, as with Aurora. When a process attempts to compute Stencil 1, the border values from Local Body 0 are automatically paged by TreadMarks into the same address space as Local Body 1. Since demand-paging is also an asynchronous operation with respect to the producer of the data, a barrier is required at the end of each time-step in the TreadMarks program. And TreadMarks is single-thread system, so there can only be one outstanding page fault.

In the MPICH implementation, the border data is exchanged between local bodies using synchronous message sends and receives. The data is placed in a local buffer that is large enough to hold both the local body and the newly received border data. The advantage of synchronous messages is that barrier synchronization is not needed after a time-step. Unlike with Aurora and TreadMarks, it is impossible for a fast processor to finish a time-step and proceed with the next time-step before the neighbouring processors are ready. For each time-step, a given processor must exchange exactly two messages, a send and a receive, with each neighbour.

Performance

A total of 32 time-steps are simulated on a 1536×1536 grid and on a 2048×2048 grid, with the performance given in Figures 5.5 and 5.6, respectively. The number of time-steps is selected to ensure a large enough execution time. The 2-D diffusion program is simpler and involves much less computation than the matrix multiplication program. For example, the granularity of work for 16 processors is substantially less than one second of real time for each simulated time-step. Each time that a barrier or synchronous message is required, performance suffers. Larger datasets and a higher granularity of work helps to improve the speedups, which is a trend that can be observed with the two datasets.

The MPICH program consistently achieves high speedups for all datasets and for all degrees of parallelism. The precise data-sharing pattern of two messages per-neighbour and per-time-step is well-suited for explicit message passing. However, small variations between when senders and receivers begin their data movement can cause delays in the exchange of data. For example, a receiver that has to wait for a sender to transmit the data can, in turn, delay the next process that is waiting for data. The cascading dependencies of synchronous messages result in a loss of performance. The non-blocking versions of the message send and receive functions were tried (and removed) in order to exploit any potential concurrency and avoid the cascading delays, but the performance was actually lower. Since standard MPICH does not support real concurrency (i.e., multiple threads), there may be no performance advantage to using non-blocking message passing for this application.

The Aurora program achieves higher performance than MPICH for 2 and 4 processors as

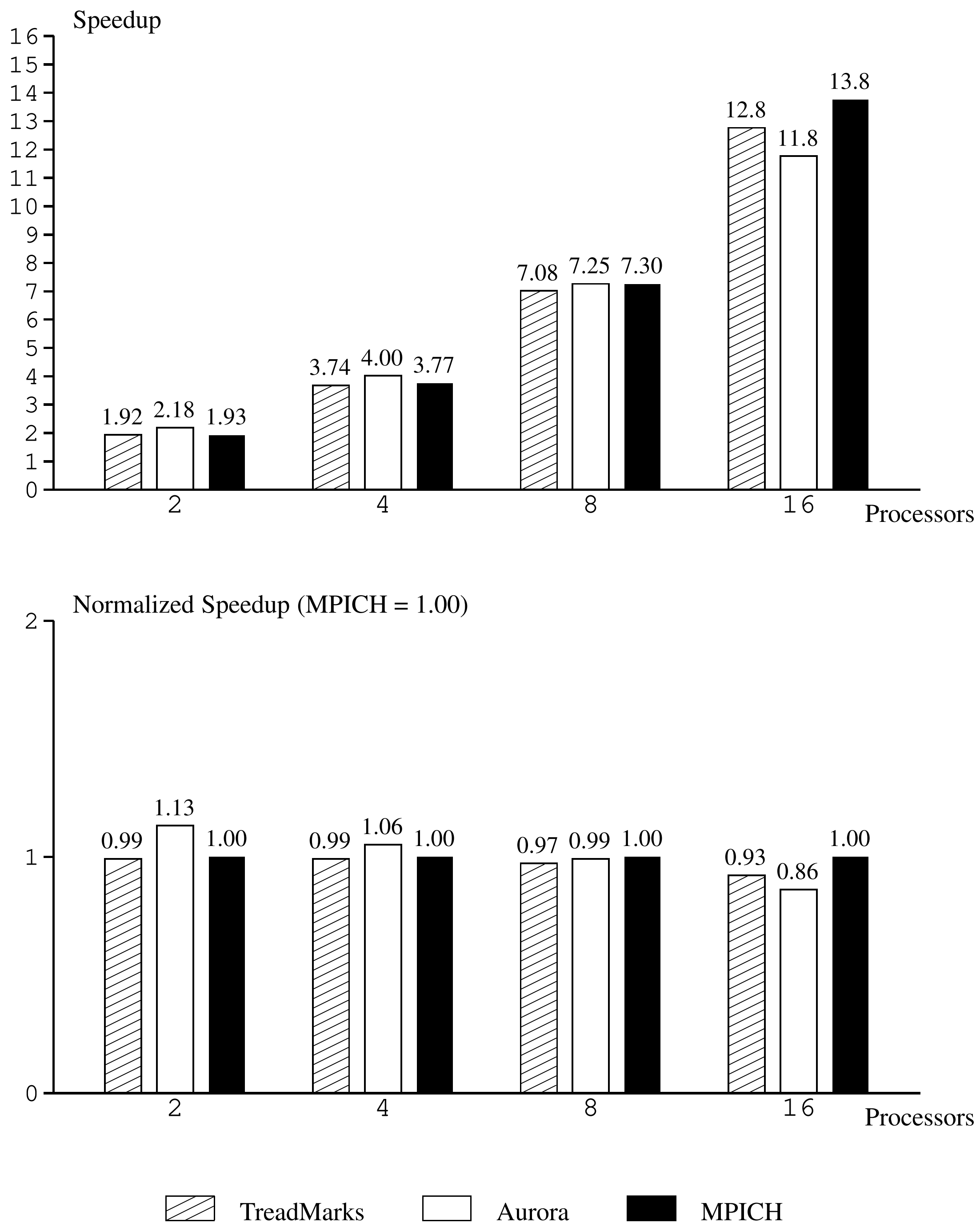


Figure 5.5: Speedups for 2-D Diffusion, 1536 × 1536, 32 timesteps

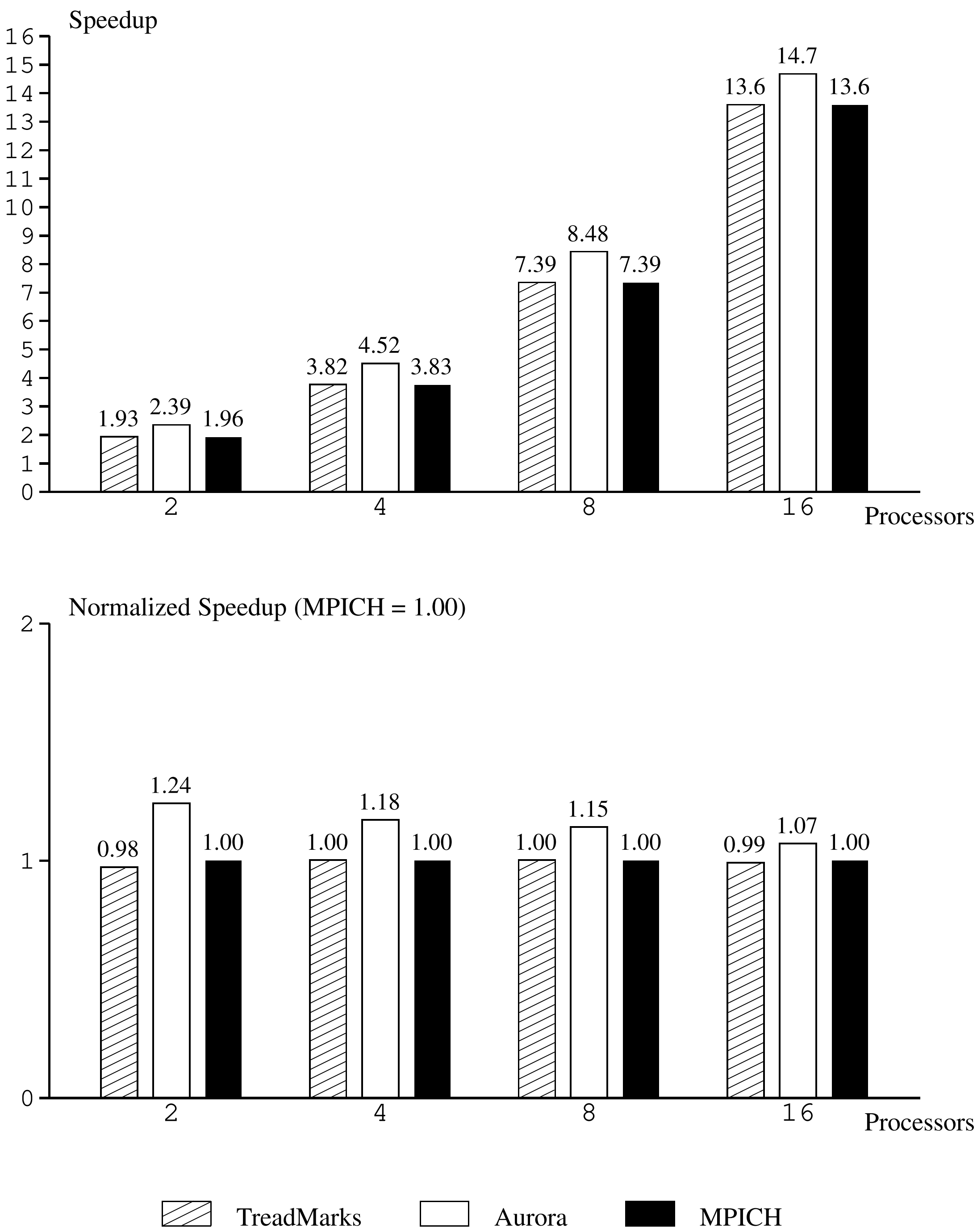


Figure 5.6: Speedups for 2-D Diffusion, 2048 × 2048, 32 timesteps

a result of the increased concurrency of a multithreaded run-time system. However, the performance advantage decreases as the degree of parallelism increases because Aurora requires a barrier after each simulated time-step and MPICH does not. As the number of processors increases, the cost of the barrier also increases for Aurora, which hurts its performance. For the 1536×1536 dataset, Aurora is 13% faster than MPICH on 2 processors, but Aurora's performance steadily declines as more processors are added until, for 16 processors, Aurora achieves only 86% of MPICH's performance. However, the real time difference between Aurora's 2.69 seconds and MPICH's 2.31 seconds for that data point is only 0.38 seconds. The granularity of work of the smaller dataset with 16 processors is very low and the performance difference can be accounted for in terms of the barriers required for Aurora and the additional overheads that Aurora adds to each message across the network. Recall that MPICH is also used as the lowest layer of Aurora's run-time system, so Aurora's cost per message is always higher than MPICH.

When there is sufficient granularity of work, such as for smaller numbers of processors or if the dataset is large enough, Aurora is able to outperform MPICH. For the 2048×2048 dataset, Aurora is 24% faster than MPICH on 2 processors and 7% faster on 16 processors. The benefits of Aurora's asynchronous and concurrent data movement appear to outweigh the barrier and per-message overheads. Presumably, if the number of processors could be increased to, say, 32 processors, Aurora's performance would once again lag MPICH since each processor would have less computation to perform. Whether Aurora or MPICH is faster depends on the granularity of work.

Aurora sometimes achieves super-linear speedups on 2, 4, and 8 processors due to hardware cache effects. Since the parallel processes only compute on a portion of the simulated grid (i.e., the local body), they benefit from having caches that are larger relative to the size of the working set than the sequential program, whose working set is the entire grid. Presumably, both TreadMarks and MPICH also benefit from the cache effects but their other communication overheads prevent them from actually achieving super-linear speedup. Although beneficial, super-linear speedups due to cache effects are not particularly noteworthy and are an orthogonal issue to data-sharing optimizations.

TreadMarks closely matches the performance of MPICH, with the speedups for the 2048×2048 dataset nearly identical between the two programs. Interestingly, TreadMarks uses a single outstanding request-response approach to handling page faults, which limits the amount of concurrency in the data transfer in a similar way as MPICH's send-receive mechanism. Since Aurora exploits more concurrency in its run-time system, TreadMarks is slower than Aurora for all the data points except when using 16 processors to solve the 1536×1536 dataset. The

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 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.

Table 5.3: Per-Context and Per-Object Optimizations in Aurora's PSRS Program

real time difference between TreadMarks and Aurora is a low 0.26 seconds for this data point and may be the result of a more efficient barrier implementation in TreadMarks.

Overall, the performance of TreadMarks, Aurora, and MPICH are quite comparable, but there are some notable trends. For low granularity data points, MPICH has a performance edge over both Aurora and TreadMarks since it does not need to perform a barrier after each time-step. A small dataset (e.g., 1536×1536) or a large number of processors (i.e., less computation per-node with 16 processes than with 2 processors), can result in a low granularity of work. But, for problems of sufficient granularity, such as the largest dataset of 2048×2048, Aurora is faster than both MPICH and TreadMarks by up to 24%, due to its multithreaded run-time system.

5.4.3 Parallel Sorting by Regular Sampling

Design and Implementation

The Parallel Sorting by Regular Sampling (PSRS) algorithm is a general-purpose and comparison-based sort with key exchange [Shi and Schaeffer, 1992, Li *et al.*, 1993]. 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.

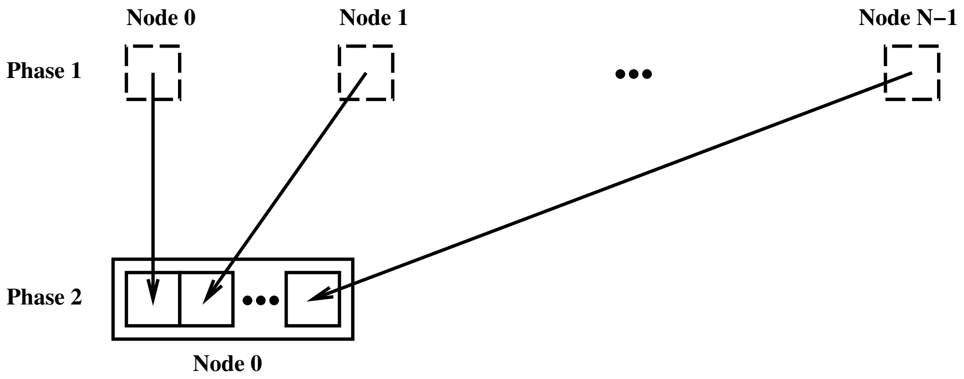
Conceptually, a multi-phase algorithm with several shared-data objects, like PSRS, is particularly well-suited for the per-context and per-object data-sharing optimizations in Aurora. The optimizations required, and the objects that are optimized, differ from one phase to another phase. Table 5.3 summarizes 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.

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 (Figure 5.7). This pattern has also been described as a gather operation [Snir *et al.*, 1996]. 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

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

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




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

Figure 5.7: Producer-Consumer Idiom in Aurora's PSRS Program

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., "all-gather") semantics and all-to-all sharing of a read cache in matrix multiplication, `distmem-cpy()` 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 [Li *et al.*, 1993]. 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. And, 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 Figure 5.8 and Figure 5.9. 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. 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 Figure 5.9 because they are too large for this data point.

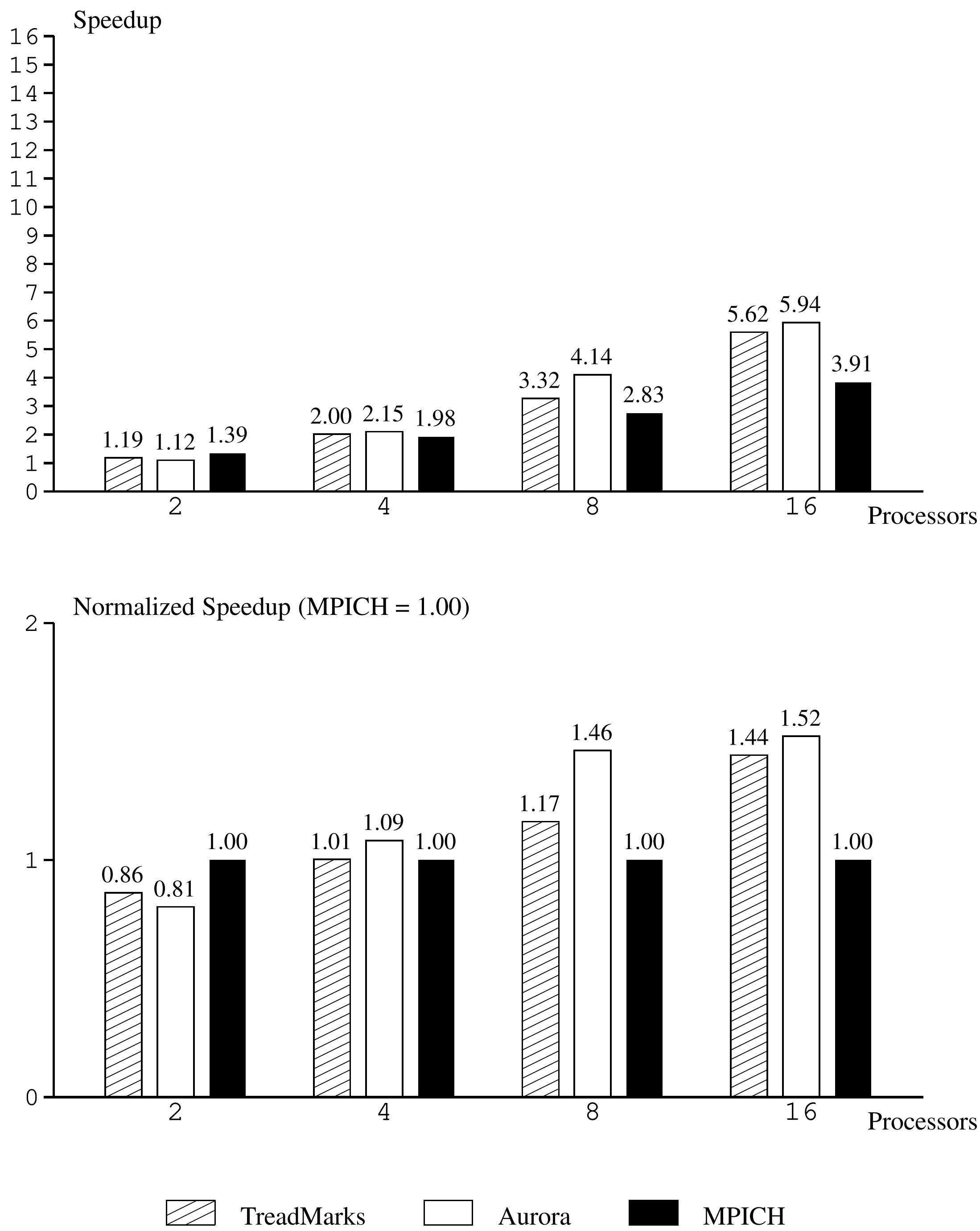


Figure 5.8: Speedups for Parallel Sorting (PSRS), 6 million keys

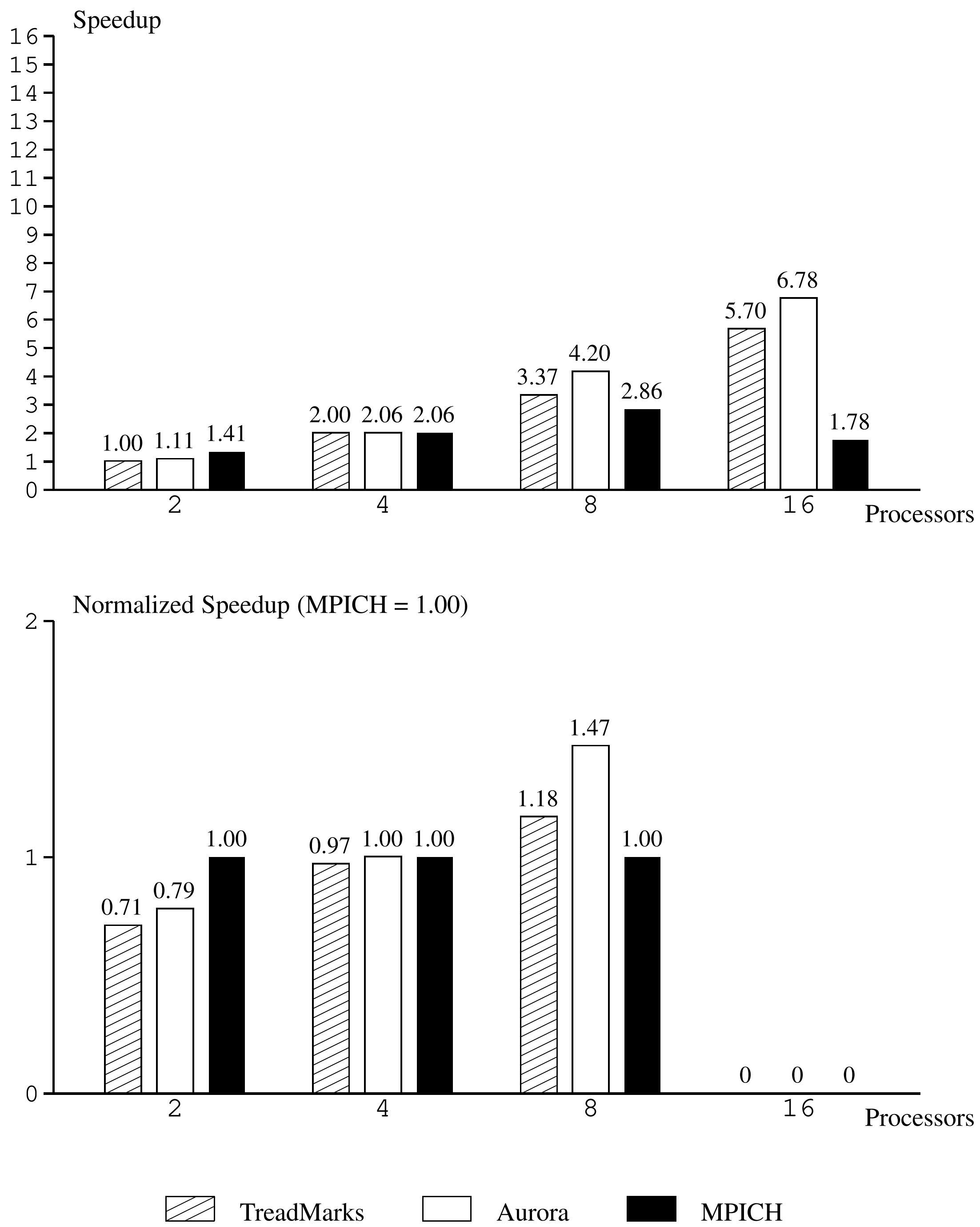


Figure 5.9: Speedups for Parallel Sorting (PSRS), 8 million keys

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.

5.4.4 Travelling Salesperson Problem

Design and Implementation

The Travelling Salesperson Problem (TSP) comes from the domain of combinatorial optimization. Given a set of cities and paths between cities with specific costs, the goal is to find a tour of minimal cost that visits all the cities exactly once. The basic problem has many variations, but we focus on the most general case in which the cities are fully connected and where the costs are not symmetric (i.e., the cost of travelling from City A to City B is not necessarily the same as from City B to City A). Recall that the original TreadMarks program and the datasets have been adopted from the TreadMarks research group at Rice University [Lu *et al.*, 1995, Amza *et al.*, 1996, Lu *et al.*, 1997].

There are two main shared data structures in the TreadMarks program. First, a priority queue is used to heuristically order the set of partial tours still under consideration. A partial tour is an ordered sequence of cities that have been visited, but not all of the cities have been included in the tour. The priority queue is implemented as a heap (i.e., a binary tree in an array). Second, the current best complete tour and its cost, which represents an upper bound on the minimal tour, is used to determine when a partial tour may be safely cut off from further consideration because it is already too expensive to be the minimal tour.

The parallel algorithm, as implemented by the Rice group, is a branch-and-bound search that uses the priority queue to select the partial tour with the lowest heuristic cost for further exploration [Lu *et al.*, 1995, Lu *et al.*, 1997]. Tours are recursively split and placed on the priority queue until they are a threshold number of cities in length. A partial tour that is within the threshold from completion is solved directly by exhaustive enumeration and forms the basic unit of work for the parallel processes. Although many TSP algorithms are based on the basic branch-and-bound framework, the algorithm used in the TreadMarks program would not be considered state-of-the-art. In particular, the heuristics for estimating the cost of a tour and for enumerating the possible tours are not particularly sophisticated. Still, we are primarily interested in the distributed data-sharing patterns and idioms that arise from this problem and the TreadMarks algorithm fulfills that purpose.

In the original TreadMarks implementation, the priority queue is placed in shared memory and a lock is used to synchronize read and write accesses to the queue. No significant changes have been made to the TreadMarks source code. All processes are equal in their ability to access and update the queue. Removing the lowest cost tour from the queue requires that the heap be updated, which can result in several updates to the tree at disparate locations (i.e., the accesses can have poor locality of reference). Furthermore, as different processes access the queue in order to find their next unit of work, the updated contents of the queue can become distributed over several processor nodes. As the lock migrates from node to node, the current TreadMarks protocol does not eagerly gather the updates at a well-known home node. Consequently, when manipulating the heap, there may be page faults on different pages of shared data in the priority queue, and it may be necessary to gather update information from multiple nodes [Lu *et al.*, 1995, Lu *et al.*, 1997].

The current best bound is also placed in shared memory and accesses are synchronized with a lock that is independent of the lock for the priority queue. Since reading and writing the best bound information requires an expensive synchronization, the TreadMarks program does not perform a lock unless a local process has found a candidate best tour. In order to verify that the new tour is really the best tour so far, a synchronization becomes necessary. The drawback of the TreadMarks approach is that new bounds values are not disseminated to all the processes in a timely manner. And, by using stale bounds data, processes may continue expanding tours that might otherwise have been eliminated because they have already been proven to be non-minimal. This is an example of the classic trade-off between the overheads of data sharing and benefits of having timely information. We return to this trade-off in the discussion below about Aurora.

The message-passing implementation is the original PVM-based program provided by the Rice research group. No significant changes have been made to the PVM-based source code. It is, of course, possible to port the code to use MPICH, but we have chosen to maintain the original code for three main reasons. First, PVM is also a well-known message-passing system with a reputation for high performance. Second, the PVM-based code has been independently-written and provides an external check against programmer-specific biases. Third, the message-passing TSP program is not communication intensive. Therefore, TSP is unlikely to reveal any significant system-specific bottlenecks or optimizations between MPICH and PVM. In fact, an inspection of the source code reveals that only the basic message send and receive functions (i.e., `pvm_send()` and `pvm_recv()`) are used during the actual computation.

The message-passing program differs from the TreadMarks program in that there is a designated process that manages the priority queue. Instead of a process individually accessing the queue to grab a new unit of work, the process sends a request message to the queue manager, which returns a unit of work in a reply message. The queue manager is the master process and the other processes are all workers. Furthermore, when a worker process requests a new unit of work, it also piggy-backs its local best tour and bound on the request message. The queue manager records the global best tour and bound and includes the global best bound with each unit of work that is sends back to requesting processes. Thus, the best tour and bound information is updated once for each unit of work in the message-passing program. By creating a specific queue manager, all of the updates to the priority queue are localized and there is never any need to gather the updates from multiple nodes, as is the case with TreadMarks. The lack of appropriate synchronization mechanisms (e.g., counting semaphores) makes it difficult to implement a master-worker strategy using TreadMarks, which is possibly why the Rice group did not attempt it.

The Aurora program follows the same overall master-worker structure of the message-passing program while still maintaining the shared-data abstraction. Aurora's support for both task and data parallelism makes this process structure possible. In fact, the initial port of the TreadMarks code to Aurora maintained the original "all processes are equal" (i.e., SPMD) structure. However, that approach was found to perform poorly, so the master-worker process structure was adopted with minimal changes to the source code. Aurora's support for data placement (i.e., its `Nodes()` interface) makes it trivial to co-locate the necessary data structures with the master, while ABC++'s support for active objects makes it easy to create the necessary process structure for a priority queue manager (i.e., master). The manager is implemented as an active object and the other processes use smart messages to communicate with it and obtain new units of work. A function call in the TreadMarks program is replaced with a smart message, which syntactically resembles the original function call.

Since the manager controls the priority queue, the queue's data structures are explicitly co-located with the manager and it accesses the data using the owners-computes scoped behaviour. The scoped behaviour is used for several different data structures, both vectors and scalars, and in several different functions. Despite the dominant use of owner-computes, the queue's data structures are still stored as `GVectors` and `GScalars` and any process can access it from any node, as per the shared-data abstraction. It is just that, for performance reasons, we have limited access to the queue manager. Unlike the previous applications, which all use built-in datatypes for shared data, Aurora's TSP uses a number of user-defined datatypes (e.g., `GVector<PrioQElement>` and `GVector<TourElement>`) further demonstrating the generality of the class templates.

As previously discussed, the timely propagation of a new upper bound on the minimal tour's cost can reduce the number of nodes searched by generating more cut-offs. The tour bound information is frequently read from, to see if the current partial tour can be cut off, and infrequently written to when a cheaper complete tour has been found. The tour bound is also strictly monotonically decreasing since tours with a larger cost than the current best bound are not significant. Therefore, the `GSReadMostly` behaviour is used for the `GScalar` that is used for the tour bound. As previously discussed, `GSReadMostly` supports the eager propagation of updated values and it also maintains the monotonically decreasing property.

This scoped behaviour has three advantages over the way TreadMarks manages the bounds information. First, the monotonic property of the scoped behaviour eliminates the need to use expensive locks. Second, new bounds are propagated eagerly to all processes to generate as many cut-offs as possible. Third, the scoped behaviour is applied per-object, so that other shared-data objects are not affected. Eager update policies have been proposed and implemented for various DSM systems, but false sharing continues to be problematic for some systems.

Performance

For the systems that we are studying, the absolute and normalized speedups are shown in Figures 5.10, 5.11, 5.12, and 5.13. Note that, since the original Rice source code for the TreadMarks and PVM programs are substantially unchanged, they do not follow the "average of five runs" timing policy. Instead of making significant changes to the source code from Rice, it was decided to leave the code unchanged. And, no significant timing variation within a job are observed in the Aurora version of TSP (i.e., range is $\leq 2\%$ of average), so the timings are expected to be stable within a job.

As with other optimization problems, the performance of TSP is particularly dependent on the specific problem instance being solved. Therefore, we use four different datasets that span a spectrum of execution times and solution complexity, as noted in Table 5.1. Each dataset, and the name of the dataset, is taken verbatim from the Rice group.

One of the reasons TSP was selected for the test suite is because its data-sharing patterns are qualitatively different from those of the other applications and because there can

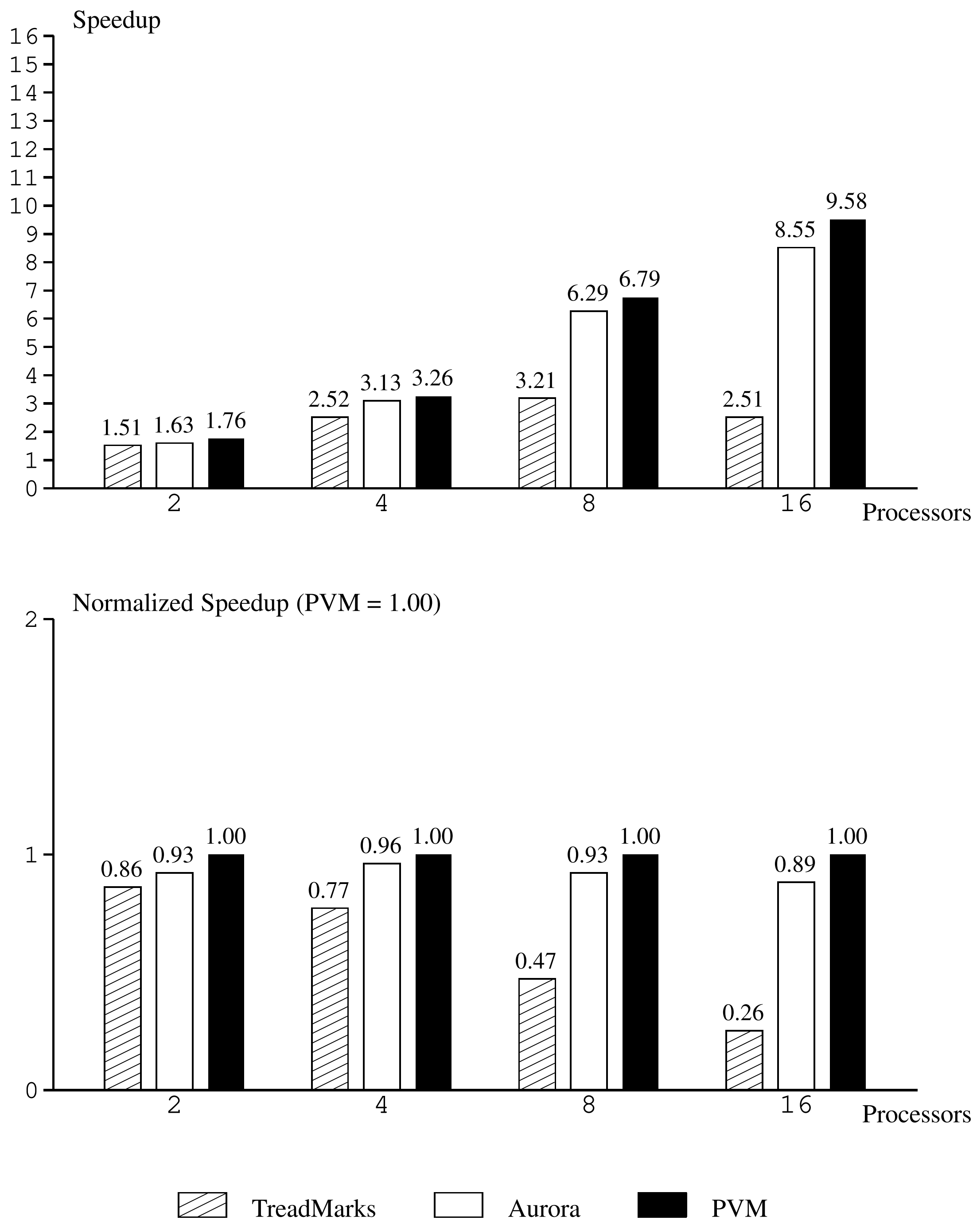


Figure 5.10: Speedups for Travelling Salesperson (TSP), Dataset 17.large

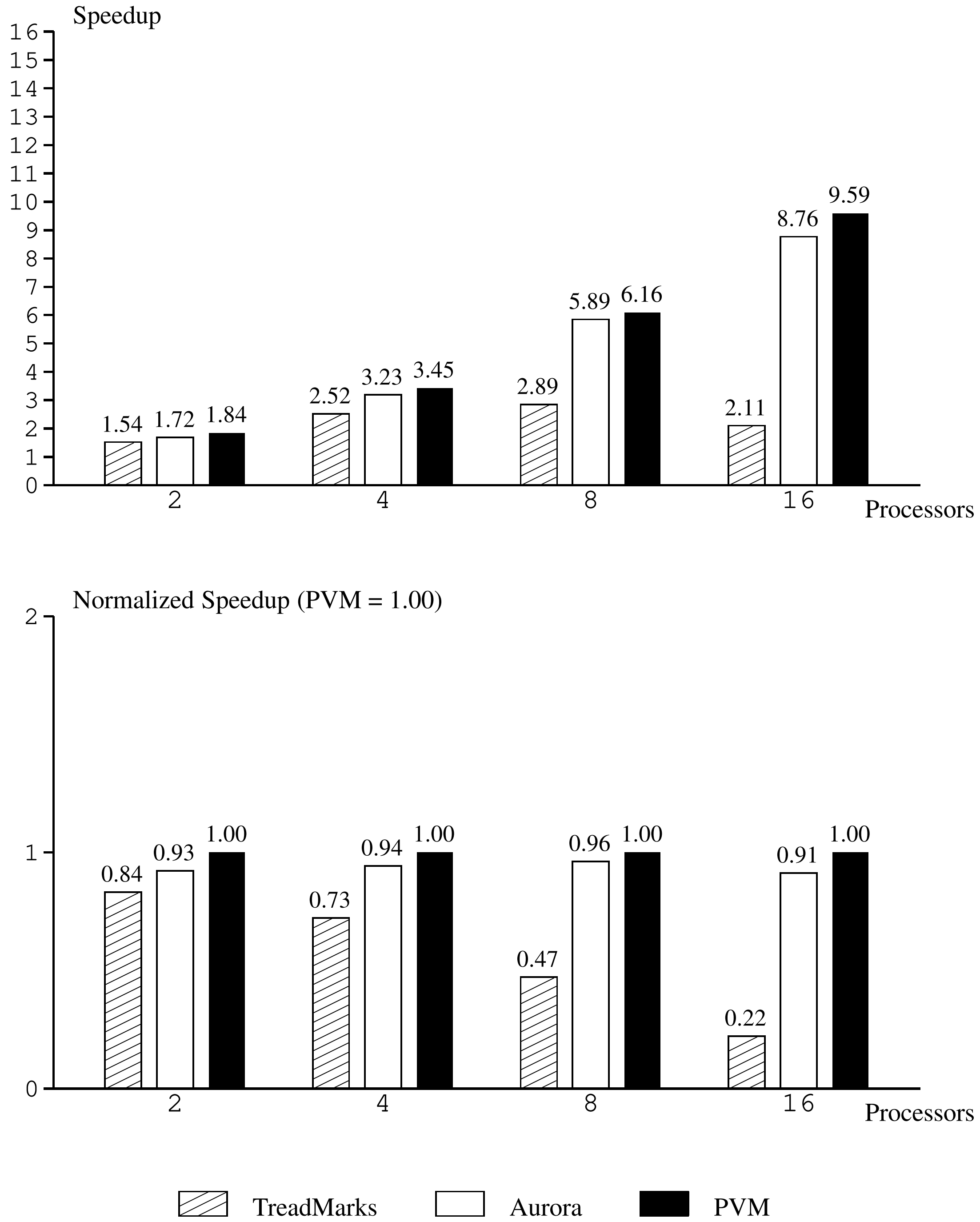


Figure 5.11: Speedups for Travelling Salesperson (TSP), Dataset 18b

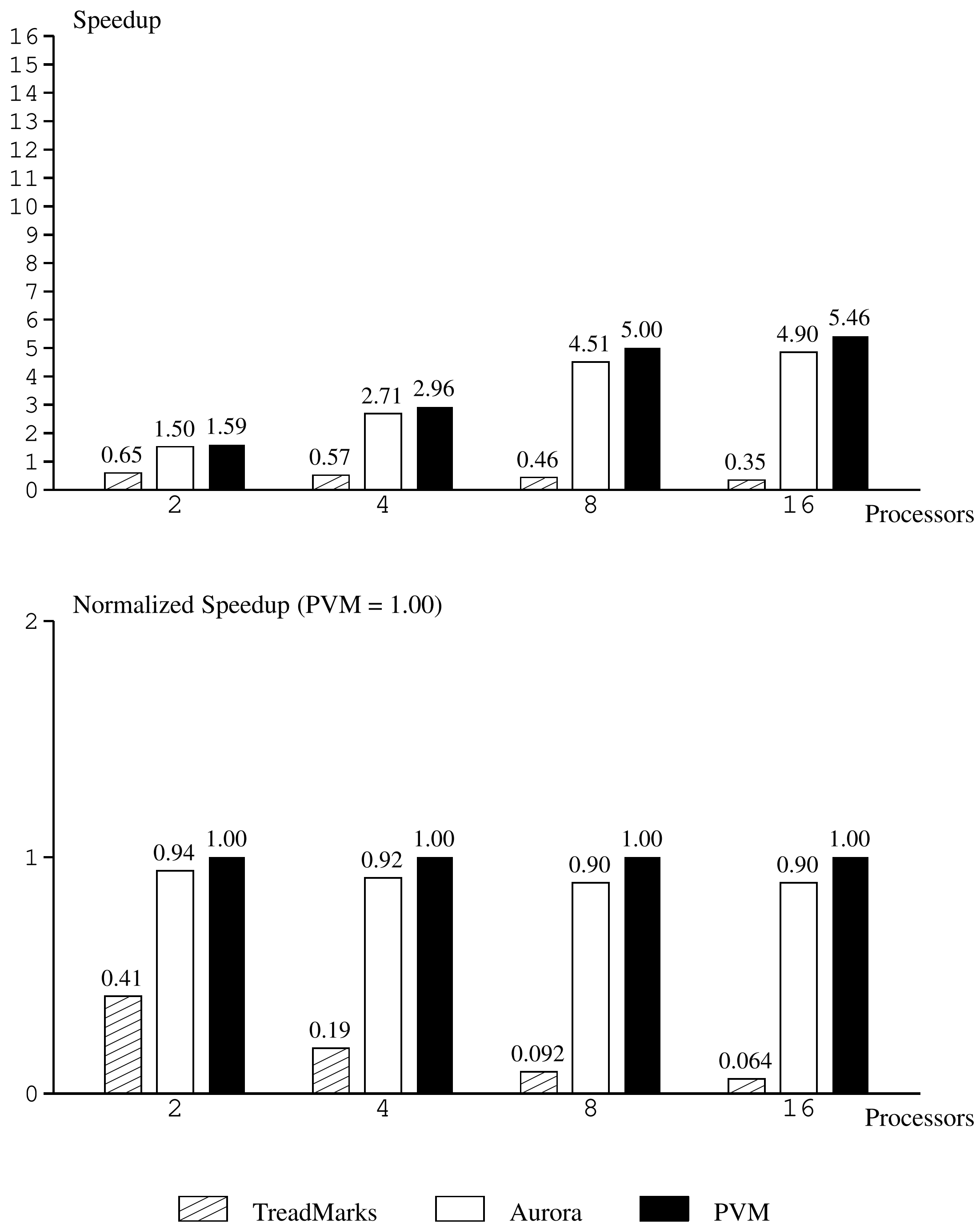


Figure 5.12: Speedups for Travelling Salesperson (TSP), Dataset 20

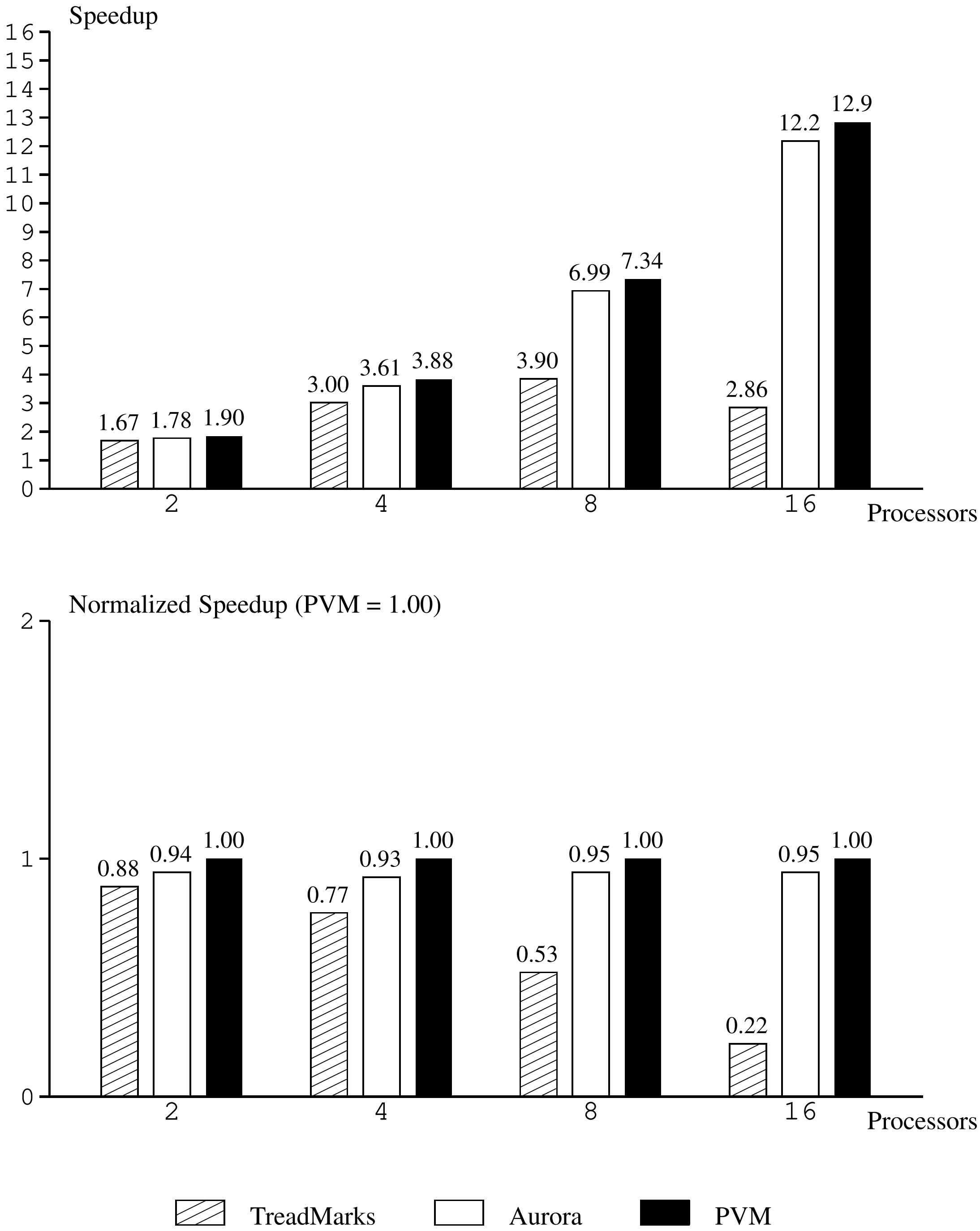


Figure 5.13: Speedups for Travelling Salesperson (TSP), Dataset 19b

be substantial quantitative differences between different TSP datasets. Note that the absolute speedups can vary considerably from one dataset to the next and, unlike the other applications, longer-running datasets do not always generate higher speedups. For example, all three systems achieve their lowest speedups with dataset "20" even though it has the second longest sequential execution time. The reason is the granularity of work: dataset "20" creates the most number of partial tours in the priority queue (4,154 tours) and, of course, each partial tour results in data sharing. For each partial tour, there is either communication between the master and worker process or communication related to critical sections for updating the priority queue. The second largest number of partial tours is 618 for dataset "19b", which is almost a factor of seven fewer. And although problems involving tours of more cities are usually more complicated than tours with fewer cities, that is not always the case. For example, dataset "19b" is a 19-city problem that requires 51.5 seconds of sequential execution time while dataset "20" is 20-city problem that only requires 38.2 seconds to solve. Not surprisingly, there is a much better correlation between the number of nodes searched and the execution time.

Also, as with all parallel branch-and-bound algorithms, the solution time has an element of non-determinism. Without a common clock, the relative progress of the worker processes can vary and affect the timeliness of when new bounds on the minimal tour are found. Since the new bounds are used to cut-off unpromising partial tours, the number of tours and nodes searched can vary from run to run. For this specific TSP algorithm and datasets, the observed variance in tours and nodes searched is low, but there is still some non-determinism.

Three observations can be made about the performance results. First, the message-passing program outperforms both Aurora and TreadMarks for all four datasets and for all the degrees of parallelism. Second, in turn, the Aurora program outperforms TreadMarks for all datasets and for all degrees of parallelism. The performance gap between message passing versus TreadMarks (and Aurora versus TreadMarks) can be substantial. Third, the performance gap between message passing over TreadMarks (and Aurora versus TreadMarks) increases as the degree of parallelism increases, which suggests a processor scalability problem with TreadMarks.

In analyzing the performance results, it is important to keep in mind that TSP is qualitatively a very different application than matrix multiplication, 2-D diffusion, or PSRS. In particular, the nature of data sharing between master and worker in the message-passing and Aurora programs requires very few bytes of data to be communicated. The representation of a partial tour and the current best bound on the minimal tour requires less than 200 bytes. In contrast to the data-intensive matrix multiplication and PSRS applications, TSP is not limited by communication bandwidth. The number of messages, which is proportional to the number of partial tours that are given out as parallel work, is a more important determinant of performance. And, by the nature of how TreadMarks maintains consistency on the heap-based priority queue, TreadMarks generates more messages for each access of the priority queue than either message passing or Aurora.

Since Aurora exists as a software layer above message passing (MPICH in the case of Aurora's run-time system), Aurora adds extra overhead to each message (i.e., smart message) between master and worker. This accounts for the persistent edge in the message-passing performance. As noted in a previous study [Lu *et al.*, 1995, Lu *et al.*, 1997], the explanation for TreadMarks's poor scalability is the large number of page faults and messages required to maintain the migratory data of the priority queue. Therefore, TreadMarks's overheads for each unit of work is much higher than either Aurora or message passing.

Aurora's worst case performance is a low of 89% of message passing's speedup for dataset "17.large" on 16 processors. Of course, dataset "17.large" is also the smallest of the four TSP datasets. In terms of real time, the difference between Aurora and message passing is only 0.16 seconds (message passing has a real time of 1.29 seconds while Aurora has a real time of 1.45 seconds), which is negligible. In contrast, TreadMarks requires 4.93 seconds to solve dataset "17.large" on 16 processors, which is a substantial difference. For all the other 16 processor data points, the real times for Aurora are within 0.80 seconds of the comparable message-passing real time. Therefore, although the conclusion is that message passing is faster than Aurora, the performance margin is small.

5.5 Summary

One of the stated goals of this research is to develop a system that supports a high-level shared-data programming abstraction and that can achieve near-message-passing levels of performance. As the results of this experiment show (Table 5.4), the Aurora system is able to match or exceed the performance of MPICH/PVM for the applications and datasets that we have studied. In some circumstances, Aurora is marginally slower than message passing and, in other circumstances, Aurora is marginally to substantially faster than message passing. In particular, Aurora's support for bulk-data transfer and asynchronous communications give it a performance edge for the matrix multiplication, 2-D diffusion, and PSRS applications, especially for 8 or more processors. And, these optimizations are made possible by the high-level semantics and integration of scoped behaviour throughout the layers of Aurora.

Aurora is also able to match or exceed the performance of TreadMarks, a well-known DSM system. The most dramatic illustration of Aurora's advantage can be seen in the results

Application	**TreadMarks**	**Aurora**	**MPICH/PVM**
Matrix Multiplication	Comparable to Aurora, but never faster. Aurora's performance advantage grows with the number of processors and as the problem size increases.	Highest performance overall, especially for 8 or more processors. Significantly faster (41% to 43% faster) than MPICH for 16 processors.	MPICH: Good performance for 8 or fewer processors. Scalability problem at 16 processors.
2-D Diffusion	Comparable to MPICH, but never faster. Always slower than Aurora except for one data point.	Better than MPICH for 4 or fewer processors. For largest dataset, Aurora is always faster than TreadMarks and MPICH.	MPICH: Consistently high speedups. Small advantage over Aurora for 8 and 16 processors for small dataset. Slower than Aurora for large dataset.
Parallel Sorting by Regular Sampling	Slower than Aurora by up to 25%. Avoids the performance problems of MPICH.	Highest performance overall. For 8 or more processors, Aurora is faster than MPICH, sometimes by a wide margin. For 4 or more processors, Aurora is always faster than TreadMarks.	MPICH: Faster than Aurora and TreadMarks for 2 processors. Comparable to Aurora and TreadMarks for 4 processors.
Travelling Salesperson (TSP)	Lowest performance, especially for 8 and 16 processors. Normalized speedups as low as 22% of PVM, with 6.4% being the worst case.	Achieves at least 89%, and usually over 90%, of the speedup of PVM.	PVM: Highest performance for all datasets and degrees of parallelism.

	TreadMarks	**Aurora**	**MPICH/PVM**
Overall Summary	Good performance for regular problems with good locality of reference (e.g., matrix multiplication). Poor performance for irregular problems with poor locality and many writers or migratory data (e.g., TSP).	Overall, comparable to (or better) than MPICH/PVM in performance. Usually faster than TreadMarks. Support for bulk-data transfer (e.g., matrix multiplication and PSRS) and asynchronous communication allows Aurora to exceed message passing's performance in several cases.	Sets baseline for comparison. Generally achieves high performance, but there are scalability problems when exchanging large amounts of data and for large numbers of processors.

Table 5.4: Performance Summary of Applications and Systems in Test Suite

for TSP. Aurora's ability to support both task and data parallelism allows for the more efficient master-worker process structure. Aurora's high-level shared-data abstraction is comparable to TreadMarks's shared-memory abstraction from the application programmer's point of view, but Aurora can often match the performance of message passing.

Chapter 6

Concluding Remarks

Distributed-memory platforms, such as a network of workstations, are attractive because of their ubiquitousness and good price-performance. However, achieving high performance is challenging because sharing data across distributed memories is expensive. Every additional network message or additional protocol processing overhead hurts performance. Fortunately, a number of performance enhancements are possible if the parallel programming system has the flexibility to optimize for the common data-sharing patterns and idioms found in parallel applications. To address the flexibility and performance issues, we have introduced the novel scoped behaviour abstraction. Scoped behaviour has been designed, implemented, and evaluated as part of the Aurora distributed shared data system.

Although the experimental results with the prototype Aurora system and applications are encouraging, there are limitations to this work and desirable future work that deserve comment. We begin with a discussion on the applicability of the scoped behaviour approach to other languages and applications. Finally, we conclude with a summary of the main research contributions.

6.1 Applicability to Other Languages

Although scoped behaviour, as prototyped in the Aurora system, does not require any language extensions or special compiler support, it does rely on specific aspects of C++ to maximize the transparency of its mechanisms. For example, scoped behaviour exploits C++'s object-oriented programming features, nested language scopes with name hiding, class templates, operator overloading, constructors, and destructors. If some of these features are missing in a different target language for scoped behaviour, it may be possible to support the basic scoped behaviour framework, but without some of the desirable transparency and syntactic sugar benefits. Since no attempt has yet been made to implement scoped behaviour in a different language, we can only (cautiously) speculate as to the issues that are raised.

For example, Java is an object-oriented language, but Java does not (currently) support nested language scopes with name hiding (i.e., identifier `i` cannot be reused for a variable inside a nested scope if `i` already exists in an outer scope), class templates, operator overloading, or destructors that are guaranteed to execute at the end of a scope. Without name hiding, it is impossible to create a Java object inside a nested scope with the same as an object outside of the scope. Thus, the compile-time benefits of the create-use-destroy framework are not available in Java. However, instead of a `NewBehaviour( vector1, GVReleaseC, int );` statement, it may be possible to use a method call, such as `vector1.newBehaviour( RELEASE_C );`, to configure the internal state of the object (e.g., set flags) to indicate that a different behaviour is wanted. No `vector1` object is created inside the scope. Depending on the internal state of the object, the methods can selectively redefine behaviour. At the end of the use-context, an explicit method call, such as `vector1.originalBehaviour();`, can reset the state of the object. The method-based `newBehaviour()` and `originalBehaviour()` approach is analogous to using `lock()` and `unlock()` around a critical section, and it suffers from the same drawbacks: it is possible to forget to unlock or reset the behaviour of the object at the appropriate point and the compiler will not detect the error. As previously discussed, the create-use-destroy approach automatically resets state and cleans up resources, but a method (or function call) approach requires the programmer to be more diligent.

Without class templates to implement container classes, one could attempt to use method-based polymorphism to support various built-in and user-defined datatypes, but that may be a challenging systems implementation problem. The lack of hands-on experience with Java and scoped behaviour makes it an open problem for future work. Without operator overloading, computations using shared-data objects cannot use the same infix notation as with built-in types, such as integers and floating point numbers. Although expressions such as `a = a + 5;`, where `a` is a shared-data object, are equivalent to `a.addTo( 5 );`, the lack of operator overloading does severely curtail the transparency benefits of concrete types and infix notation. And, the lack of destructors that are guaranteed to be called at the end of the language scope means that the Java programmer must explicitly add method calls, such as `vector1.originalBehaviour();` (discussed above) and `vector1.flushUpdates();`, at the end of a use-context. Overall, it would be a challenge to implement a transparent form of scoped behaviour in Java.

For non-object-oriented languages, such as C and Fortran, the challenges are even greater. The basic premise of creating abstract data types to hide the existence of distributed memories relies on language support for abstract data types. Before the invention of C++, people programmed in an object-oriented style using C's function pointers, but that often resulted in code that was difficult to understand and is not recommended. With the evolution of languages, such as Fortran, to include object-oriented features, there is still the possibility that scoped behaviour can be implemented. However, as we saw with Java, having some object-oriented language features but no class templates, no operator overloading, and no destructors can mean a large compromise on scoped behaviour's transparency. The basic ideas of scoped behaviour are portable, but the transparency aspect is also highly desirable.

Although we have discussed some of the issues regarding the applicability of scoped behaviour to languages other than C++, it should be emphasized that without an actual implementation, there may be many unknown pitfalls and windfalls. If programming languages, including Java, continue to evolve, an area of future work would be to implement scoped behaviour for a different language than C++. But, for now, the lack of certain language features makes that a less attractive research direction.

6.2 Applicability to Other Applications

The current set of Aurora applications are limited to simple scalar, vector, and matrix data structures. Of course, a number of graph-based and dynamically-allocated data structures rely on the use of pointers to link different components of the larger structure. For example, linked lists, various tree structures, and some graph theory algorithms require pointers in their data structures. Since pointers are not necessarily valid across distributed memories, a different abstraction is required in a DSD system such as Aurora.

One possible solution would be to use the handle objects (of the handle-body composite objects used in scoped behaviour) in place of pointers. Both handles and pointers can name data structures in a global namespace, they can be dereferenced to access the target data, and they can be copied. It is not a trivial conversion process to replace pointers with handles, but the two abstractions are compatible. The main disadvantage of handles over pointers is that handles require more storage. Currently, handles for scalar objects in Aurora require approximately three times more storage than a simple pointer. If the storage overheads are not an issue, then substituting handles for pointers may be a workable solution.

However, more significantly, new scoped behaviour optimizations have to be developed for data structures containing links that are handles. All of the current optimizations are designed for dense and contiguous data structures, such as vectors and matrices. The design, implementation, and evaluation of this new category of scoped behaviours would be a non-trivial extension on the current work with Aurora. Although important data structures and algorithms are currently excluded by the omission of these optimizations, the proof of concept of scoped behaviour with vectors and matrices does make it more attractive to pursue this line of research as future work. And, fortunately, any new scoped behaviour optimizations would be orthogonal to the current set of optimizations (i.e., they are added to the class library), and they can be applied on a per-object and per-context basis along with the existing optimizations.

N-body codes are part of a class of scientific applications that has gained research attention [Singh *et al.*, 1995]. The Barnes-Hut application from the SPLASH/SPLASH-2 suite [Singh *et al.*, 1992, Woo *et al.*, 1995] is an example of an N-body application. As implemented in TreadMarks, the main data structures are vectors of C structures representing "bodies" and "cells". Since Aurora already supports shared vectors of arbitrary structures (as demonstrated with `PrioQElement` and `TourElement` in the TSP application), it may be possible to use `GVectors` of these application-specific structures. However, high performance within the scoped behaviour approach requires more than just the ability to create shared-data objects. An important premise of scoped behaviour is that the programmer must be knowledgeable about the data-sharing patterns of the application and a set of optimizations must be targeted at those patterns.

Since the data-sharing patterns in N-body are distinct from other applications and complex [Singh *et al.*, 1995], the current library of scoped behaviours are not well-suited to N-body applications. For example, the TreadMarks group observes that although "the set of bodies owned by a processor are adjacent in the Barnes-Hut tree, they are not adjacent in memory" (page 74, [Lu *et al.*, 1997]). Such a data-sharing pattern exhibits good locality at a fine-grained cache line granularity (i.e., a single body or cell structure fits well within a cache line), but it is poor locality at page-based granularity (i.e., poor spatial locality between bodies and cells) [Lu *et al.*, 1997]. New optimizations must be implemented in Aurora to support the fine-grained sharing exhibited by Barnes-Hut since that kind of data sharing does not exist in the other applications. It is also difficult to speculate on what form those optimizations might take until after a more thorough study of the N-body codes. Therefore, the problem of supporting N-body codes in Aurora remains an open question.

Admittedly, the relatively small set of existing scoped behaviour optimizations does limit the range of applications that can be immediately addressed by Aurora. As new application domains are explored, new optimizations may have to be developed. But, as previously discussed, the strength of the scoped behaviour approach is that once an optimization has been implemented, it can be readily reused (where applicable) with per-object and per-context flexibility.

6.3 Research Contributions

The research contributions of the work on scoped behaviour and Aurora include:

1. We have shown how scoped behaviour provides a high degree of flexibility in optimizing shared-data objects on a per-object and per-context basis.

   By applying a set of system-provided scoped behaviours to different objects and in different phases of a program, the application programmer can implement a large number of optimization strategies. Each phase of the program and each shared-data object can be optimized in a different way.

   Existing parallel programming systems are more limited in their set of supported data-sharing optimizations and in their ability to combine optimizations within a single multi-phase parallel program.

2. We have shown how scoped behaviour can be implemented in standard C++ without any language extensions and without special compiler support.

   The create-use-destroy implementation framework of scoped behaviour, which is based on objects and language scopes, is also innovative in its ability to exploit both compile-time and run-time information for optimizations.

   By exploiting the extensive support for abstract data types in C++ and object-oriented design principles, we have implemented scoped behaviour entirely within a class library and run-time system. In contrast, some other parallel programming systems have depended upon non-standard extensions to the base language, special compilers, or special operating system support. Consequently, the engineering effort required to implement and extend those parallel programming systems tends to be significantly greater than for Aurora.

3. We have shown how the Aurora parallel programming system can achieve a level of performance that is as high, or substantially higher, than other state-of-the-art systems.

   In a performance comparison between Aurora, TreadMarks, and message passing (either MPICH or PVM) using four applications, Aurora (with only a few exceptions) is comparable in performance or faster than the other systems. In some specific cases, Aurora is several-fold faster due to data-sharing optimizations that have been made possible through the abstract data type and scoped behaviour approach.

A parallel programming system can be evaluated according to three main criteria: ease-of-use, expressive power, and performance. It is beyond the scope of this work to evaluate the ease-of-use of different systems, but it is generally acknowledged that the higher the level of abstraction, the easier it is for the application programmer to use the system. On this point, DSM and DSD systems are often easier to use than message-passing systems.

However, systems with high-level abstractions can be limited in their expressive power. Although high-level programming systems make is easier to create correct and working programs, they often do not allow much freedom for the programmer to incrementally and substantially tune for performance. Expressive power and performance are often closely linked. For example, TreadMarks emulates hardware-based shared-memory so faithfully that many programs originally developed for parallel computers can be ported to TreadMarks with a minimum of effort. However, once ported, TreadMarks does not allow the programmer to experiment with different data-sharing policies and with important process structures, such as the master-worker structure in the TSP application, with important performance consequences.

In contrast, Aurora uses scoped behaviour to support different data-sharing policies. Scoped behaviour provides both expressive power and safety because the programmer can experiment with different behaviours and different optimization strategies with a minimum number of changes to the source code. And, Aurora's expressive power is enhanced by its support for several different process models and structures, including master-worker, in the same program. By integrating the high-level shared-data objects and scoped behaviour across the various layers of software, Aurora can achieve a level of performance that compares well with other parallel programming systems.

Appendix A

Summary of the Aurora System

This is reference-oriented summary of the main features of the Aurora system. Many of the ideas are discussed in depth in the main chapters, especially in Chapter 3.

1. **Initialization:**

   Aurora programs generally have the following in their `main()` or main source code file:

   ```
   #include "aurora.H"

   Pinit( argc, argv );           // Initializes ABC++
   AuroraInit( &argc, &argv );    // Initializes Aurora
   ```

   Note that individual shared-data objects and scoped behaviours are `#include`'d via a `.H` file of the same name.

2. **Finalization:**

   Before exiting `main()`, the underlying ABC++ system requires:

   ```
   Pexit( 0 );
   ```

3. **Basic Shared-Data Objects:**

   ```
   GScalar<TYPE>
   GVector<TYPE>
   GSemaphore<int>
   ```

   Example 1: `GVector<int> vector1( 1024 );` creates a statically-sized shared vector of 1,024 integers. The elements are block distributed across all the nodes of the computation, which is known at run-time. Currently, only block-distributed data is supported.

Example 2: `GVector<int> vector1( 1024, Nodes( 0 ) );` creates a statically-sized shared vector of 1,024 integers. However, all of the integer elements are explicitly placed on Node 0.

Example 3: `GVector<int> vector1;` creates a handle to a shared vector, but the size and distribution is not yet specified. Before it can be used, the vector must be properly constructed with, say, `vector1.constructNow( vsize, 5 );` which creates `vsize` elements that are distributed across 5 nodes.

4. **Scoped Behaviours for Scalars:**

   Scoped behaviours are applied to scalars using the system-provided macro `NewBehaviourScalar`. For example:

   ```
   NewBehaviourScalar( GS_Count, GSOwnerComputes, int );
   ```

   changes the behaviour of shared integer `GS_Count` to owner-computes.

   The scoped behaviours include:

   ```
   GSOwnerComputes<TYPE>
   ```

   only allows the thread co-located with the data to access the shared scalar.

   If a scalar is only seldom updated, then reads can be made from a local cache and updates can be made eagerly by selecting scoped behaviour:

   ```
   GSReadMostly<TYPE>
   ```

   This behaviour can also be selected with a method call, `setEagerUpdate()`, for use in the global scope.

5. **Scoped Behaviours for Vectors:**

   Scoped behaviours are applied to vectors using the system-provided macro `NewBehaviour`. For example:

   ```
   NewBehaviour( mA, GVOwnerComputes, int );
   ```

changes the behaviour of shared vector `mA` to owner-computes.

The scoped behaviours include:

```
GVOwnerComputes<TYPE>
```

This scoped behaviour only allows threads that are co-located with a local portion of the shared vector to access that shared data. Method `doParallel( TEAM );` initializes the behaviour with respect to a given SPMD team. Methods `begin()`, `end()`, and `step()` iterate over the local block of data.

```
// Create a read cache and load automatically
GVReadCache<TYPE>

// Create buffers and buffer updates
GVReleaseC<TYPE>

// Support block-distributed data with padding
GVBlockedComputes<TYPE>    // Used in 2-D Diffusion
```

Method `fetchPadding()` of `GVBlockedComputes` fills the padding with up-to-date data.

```
GVReadCacheSparse<TYPE>    // Used in PSRS
```

This behaviour supports caching for reads, but it does not automatically load the cache. It does allow explicit and sparse loading of the cache.

6. **Support for SPMD-style Data Parallelism:**

   Class `Team` creates an object which is a descriptor to a team of threads than can execute in SPMD fashion. Consider:

```
// Create a team of threads
Team theTeamObject;

// Create a team of 5 threads
Team theTeamObject( 5 );

// Create a team with one thread co-located with each
// local body of shared-data object (GVector) MatrixA
Team theTeamObject( MatrixA.nodes() );
```

Once a team is created, they begin executing a function in SPMD fashion with:

```
theTeamObject.parallel( FUNCTION_NAME );
```

and the team can be finalized with:

```
theTeamObject.waitForDone();
```

When executing in SPMD fashion, the following control structure is available:

```
MASTER
{
    // Code to be executed by Node 0 only
}
END_MASTER;
```

and the team of threads can be silently synchronized with a barrier, or synchronized with debugging output sent to `stdout`, as follows:

```
// Silently synchronize
BARRIER;

// Named barrier:  String "Have reached this point"
// is sent to stdout after the barrier is executed.
N_BARRIER( "Have reached this point" );
```

Methods of class `Team` include: `getId()` which returns the identification number of the current thread and `IamDone()` which indicates that the current thread is about to exit.

7. **Auxiliary Functions:**

```
distmemcpy();
```

is analogous to `memcpy()` but works with shared vectors.

8. **Auxiliary Classes:**

   Two-dimensional matrices can be mapped to one-dimensional vectors with the help of index objects:

```
// For 2-D matrices
Index2D<TYPE> idx;

// For 2-D matrices with border padding and
// are block distributed
Index2DPad<TYPE> idxPad;
```

   After being initialized with the dimensions of the matrix (e.g., `idx.set( 10, 10 );` configures the indexing for a 10×10 matrix), then `idx.i( i, j )` returns the one-dimensional index for the `i`th row and `j`th column of the two-dimensional matrix. For a given one-dimensional index `i`, `idx.m( i )` returns the two-dimensional row number.

   For the case with padding, `idxPad.ii( i, j )` returns the same one-dimensional index, but with the padding and block distribution taken into consideration.

   In order to support shared-data objects in the global scope, an Aurora program must have the following in the global scope:

```
#include "oracle.H"
Oracle AuroraOracle;
```

9. **Implementation of Aurora:**

   Total lines of code: Approximately 16,000, mostly in the form of class templates.

   Major libraries that are linked with: ABC++, Pthreads, MPICH

Appendix B

Aurora Programs

The following segments of code have been partially cleaned up for presentation. They are included to give the reader a flavour of the programming system; a lot of the debugging and timing code have been removed. However, it should be noted that the code and the programming style remain raw since, as the saying goes: you never finish a program, you only stop working on it.

B.1 Matrix Multiplication

```
/*
======================================================================

        Matrix Multiplication Using Aurora

======================================================================
*/

#include "aurora.H"

#include "GScalar.H"
#include "GVector.H"

#include "GVOwnerComputes.H"            // Scoped behaviours
#include "GVReadCache.H"
#include "GVReleaseC.H"

#include "team.H"                       // For SPMD team
#include "oracle.H"                     // For initialization

#include <stdio.h>
#include <memory.h>                     // For memcpy()

#include "multidim.H"                   // For 2-D index object:  Index2D<>

#define NUM_RUNS        6

Oracle AuroraOracle;

Index2D<int>    idx;

/*

```

```
        Globals

*/
GScalar<int>    XnumPE;                 // Globals to pass info to all PEs
GScalar<int>    XmatrixSize;

GVector<int>    MatrixA;
GVector<int>    MatrixB;
GVector<int>    MatrixC;

GScalar<int>    RunNum;                 // Current run number

TimeObj ParTimer    ( "Total time in PARALLEL:" );

double RunTimes[ NUM_RUNS ];            // Save run times for averaging

// NOTE:  Only Node 0 calls setUp_NodeZeroOnly()
extern void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp );
extern void * matrixmultiply( void * arg );
extern void mmultiply( int * a, int * b, int * c, int m, int n, int p );

template< class C_Data >
void createData( ProcessTeam * myTeam, GVector<C_Data> & theMatrix,
                 const int randSeed, char * msg );


// Common function to parallel and sequential
inline int dotProduct( int * a, int * b, const int j, const int n )
{
        register int sum = 0, k;

        for( k = 0; k < n; k++ )
        {
                sum += a[ k ] * b[ idx.i( k, j ) ];
        }

        return( sum );
} /* dotProduct */


// SEQUENTIAL
/* mmultiply:  Multiplies matrices in parameters. */

void mmultiply( int * a, int * b, int * c, int m, int n, int p )
{
        int     i, j, k;                /* Loop indices */

        for( i = 0; i < m; i++ )
        {
                for( j = 0; j < p; j++ )
                {
                        c[ idx.i( i,  j ) ] =
                                dotProduct( &a[ idx.i( i, 0 ) ], b,  j, n );
                }
        }
} /* mmultiply */


// PARALLEL
template< class C_GV >
void Aurorammultiply(   ProcessTeam * myTeam,
                        C_GV a, C_GV b, C_GV c,
                        const int m, const int n, const int p )
{
        int     i, j, k;                /* Loop indices */

  { // New scope
```

```

        NewBehaviour( a, GVOwnerComputes, int );
        NewBehaviour( b, GVReadCache, int );
        NewBehaviour( c, GVOwnerComputes, int );

    while( a.doParallel( myTeam ) )
    {
        int * theA;

        for( i = idx.m( a.begin() ); i < idx.m( a.end() ); i += a.step() )
        {
                for( j = 0; j < p; j++ )
                {
                        theA = (int*)&( a[ idx.i( i, 0 ) ] );

                        c[ idx.i( i, j ) ] = dotProduct( theA, (int*)b, j, n );
                }
        }
    } // while

  } // End scope

} /* Aurorammultiply */


/*
------------------------------

NAME:   Main

------------------------------
*/

int main( int argc, char ** argv, char ** envp )
{
        int     i;

        // NOTE:  Everybody calls Pinit(), but only Node 0 returns from it
        Pinit( argc, argv );

        // NOTE:  Only Node 0 ever calls AuroraInit()
        AuroraInit( &argc, &argv );

        // Create the team for owner-computing on block-distributed matrix

        Team team( MatrixA.nodes() );

        team.parallel( matrixmultiply );

        printf( "Back from parallel section...\n" );

        team.waitForDone();

        printf( "About to Pexit()...\n" );

        Pexit( 0 );

        return 0;
} // main



void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp )
{
        int     i, j, k;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */
```

```

        /* Process command line */

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n----------------------------------------\n" );
        printf( "Matrix Multiplication Using Aurora\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- How large of a matrix? */

        XmatrixSize = atoi( argv[ 2 ] );  /* Get from input line: arg 2 */
        printf( "\tSquare matrix size is:  %d x %d\n",
                (int)XmatrixSize, (int)XmatrixSize );

        /* Set up indexing object */
        idx.set( (int)XmatrixSize, (int)XmatrixSize );

        /* Everybody must call method owners().  It sets-up internal values.
                Repeated SPMD in matrixmultiply() */
        idx.owners( (int)XnumPE );

        /* Allocate space for the data */
        MatrixA.constructNow( idx.size(), (int)XnumPE );
        MatrixB.constructNow( idx.size(), (int)XnumPE );
        MatrixC.constructNow( idx.size(), (int)XnumPE );

} /* setUp_NodeZeroOnly */




void * matrixmultiply( void * arg )
{
        ProcessTeam * myTeam;
        myTeam = AuroraInitThread( arg );

        int     i, j, k;
        int     m, n, p;                /* Matrix dimensions */
        int     start, end;


        // NOTE:  Only setUp_NodeZeroOnly() initialized (some of) these
        //      values, so we repeat them here.
        m = n = p = (int)XmatrixSize;
        idx.set( (int)XmatrixSize, (int)XmatrixSize );

        // Repeat this here in Aurora version because previous call was
        //      only by the master
        // Everybody must call method owners().  It sets-up internal values.
        idx.owners( (int)XnumPE );


BARRIER;                /* Wait for all PEs to start-up */

N_BARRIER( "Start All Runs" );

        /* Do NUM_RUNS number of test, and average statistics */

        for( RunNum = 0; RunNum < NUM_RUNS; )
        {
                /* Master will increment RunNum later */

                /* Create data in parallel */
```

```

N_BARRIER( "Start Create Data: A" );
                createData( myTeam, MatrixA, Seed[ RunNum ],
                        "MatrixA Partition" );

N_BARRIER( "Start Create Data: B" );
                // Note:  multiplicative factor of 7
                createData( myTeam, MatrixB, 7 * Seed[ RunNum ],
                        "MatrixB Partition" );

N_BARRIER( "End Create Data" ); // Wait for data to be created

                /*
                        ----- PARALLEL -----
                */
                if ( (int)XnumPE > 1 )
                {
N_BARRIER( "Start Parallel : C = A x B" );

                    { TimeScope timer( ParTimer, false );

                        Aurorammultiply( myTeam,
                                MatrixA, MatrixB, MatrixC, m, n, p );

N_BARRIER( "End C = A x B : Start B = A x C" );

                        Aurorammultiply( myTeam,
                                MatrixA, MatrixC, MatrixB, m, n, p );

N_BARRIER( "End B = A x C" );

                    } // End timer
                }
BARRIER;

MASTER
{
                /* Done multiplying.  Calc statistics. */
                printf( "Done!  Run Number %d.\n", (int)RunNum );

                ParTimer.print( "\n" );
                if( (int)RunNum > 0 )
                        ParTimer.addSample();

} // MASTER
END_MASTER;

BARRIER;

MASTER
{
                RunNum++;                       /* Ready to do next run */
} // MASTER
END_MASTER;

                fflush( stdout );
N_BARRIER( "End RunNum" );
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );

MASTER
{
        printf( "\n" );
        printf( "%d PEs / %d x %d / Aurora Version of MM2\n\n",
                (int)XnumPE, (int)XmatrixSize, (int)XmatrixSize );
```

```

        ParTimer.stats( "Avg time (FINAL)\n" );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        myTeam->IamDone();

        return NULL;
} /* matrixmultiply */



// Create data for my co-located data via owners-computes
//      The values are random (via srand()) and the seed if a function
//      of the parameter 'randSeed' and my rank in my team.
template< class C_Data >
void createData( ProcessTeam * myTeam,
                 GVector<C_Data> & theMatrix,
                 const int randSeed,
                 char * msg
)
{
    { // New scope
        NewBehaviour( theMatrix, GVOwnerComputes, int );

        while( theMatrix.doParallel( myTeam ) )
        {
                int i = myTeam->getId();

                int start = theMatrix.begin();
                int end   = theMatrix.end();

                /* One-beyond (instead of "end - 1") semantics now */

                /* Fill [ start , end ) with data */
                srand( ( i + 1 ) * randSeed );

                int theVal, k;
                for( k = start; k < end; k++ )
                {
                        theVal = rand();        /* Create data */
                        theVal %= 31;   /* Limit the range to limit overflow */

                        theMatrix[ k ] = theVal;
                }
         } // doParallel
    } // End scope
} /* createData */
```

B.2 2-D Diffusion

```
/*
======================================================================

        2D Diffusion Using Aurora

======================================================================
*/

#include "aurora.H"

#include "GScalar.H"
#include "GVector.H"

#include "GVBlockedComputes.H"  // Scoped behaviour

#include "team.H"               // For Team
#include "oracle.H"

#include <stdio.h>
#include <memory.h>             // For memcpy()

#include "multidim.H"           // For class Index2DPad<>

#define NUM_RUNS        6

Oracle AuroraOracle;

Index2DPad<float> idx;

/*

        Globals

*/
GScalar<int>    XnumPE;         // Globals to pass info to all PEs
GScalar<int>    XmatrixSize;

int MaxIter = 32;
const float InitVal = 100000.0;
float TotalMass;
int StartMassRow;
int EndMassRow;
int MidPointRowCol;

GVector<float>  MatrixA;
GVector<float>  MatrixB;

GScalar<int> RunNum;            // Current run number

TimeObj TotalTimer( "Total time            :" );

// NOTE:  Only Node 0 calls setUp_NodeZeroOnly()
extern void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp );
extern void * doRuns( void * arg );
extern void Diffusion2D( float * MatrixACopy, float * MatrixBCopy );


// Shared by parallel and sequential
inline void blockStencil(
        const int rowStart, const int rowEnd, const int step,
        const int size,
        float * mm, float * ans,
        Index2DPad<float> & id )
{
        int     i, j;
```

```

        // For each row
        for( i = rowStart; i < rowEnd; i += step )
        {
                // Avoid edges
                if( ( i == 0 ) || i == ( size - 1 ) )
                        continue;

                // For each column
                for( j = 0; j < size; j++ )
                {
                        // Avoid edges
                        if( ( j == 0 ) || j == ( size - 1 ) )
                                continue;

                        float newVal =  (
    (float)
    mm[ id.ii( i-1, j-1 ) ] + mm[ id.ii( i-1, j ) ] + mm[ id.ii( i-1, j+1 ) ] +
    mm[ id.ii( i  , j-1 ) ] + mm[ id.ii( i  , j ) ] + mm[ id.ii( i  , j+1 ) ] +
    mm[ id.ii( i+1, j-1 ) ] + mm[ id.ii( i+1, j ) ] + mm[ id.ii( i+1, j+1 ) ]

                                        ) / 9.0;

                        ans[ id.ii( i, j ) ] = newVal;
                }
        }
} // blockStencil




template< class C_GV >
void AuroraDiffusion2D( ProcessTeam * myTeam, C_GV mA, C_GV mB )
{
        int     i, j, k;
        int size = (int)XmatrixSize;

        Index2DPad<float> myIdx;

        myIdx.set( size, size );

        // For each iteration
        for( int iter = 0; iter < ( MaxIter / 2 ); iter++ )
        {
            { // New scope
                NewBehaviour( mA, GVBlockedComputes, float );
                NewBehaviour( mB, GVBlockedComputes, float );

                mA.doParallel( myTeam );                // Initialize
                mB.doParallel( myTeam );                // Initialize

                int mab = idx.m( mA.begin() );
                int mae = idx.m( mA.end() );
                int mas = mA.step();

                mA.fetchPadding( myTeam );

                blockStencil( mab, mae, mas, size,
                        /* source -- timePrevious */ (float *)mA,
                        /* target -- timeNow      */ (float *)mB,
                              myIdx );

            } // End scope

N_BARRIER( "End Compute B" );

            { // New scope
```

```
                NewBehaviour( mA, GVBlockedComputes, float );
                NewBehaviour( mB, GVBlockedComputes, float );

                mA.doParallel( myTeam );                // Initialize
                mB.doParallel( myTeam );                // Initialize

                int mbb = idx.m( mB.begin() );
                int mbe = idx.m( mB.end() );
                int mbs = mB.step();

                mB.fetchPadding( myTeam );

                blockStencil( mbb, mbe, mbs, size,
                        /* source -- timePrevious */ (float *)mB,
                        /* target -- timeNow      */ (float *)mA,
                              myIdx );
            } // End scope

N_BARRIER( "End Compute A" );

        } // iteration

} // AuroraDiffusion2D



/*
------------------------------

NAME:   Main

------------------------------
*/

int main( int argc, char ** argv, char ** envp )
{
        int     i;

        // NOTE:  Everybody calls Pinit(), but only Node 0 returns from it
        Pinit( argc, argv );

        // NOTE:  Only Node 0 ever calls AuroraInit()
        AuroraInit( &argc, &argv );

        // NOTE:  Only Node 0 calls setUp_NodeZeroOnly()
        setUp_NodeZeroOnly( argc, argv, envp );

        // Create the team for owner-computing on Xarray

        Team team( MatrixA.nodes() );

        team.parallel( doRuns );

        printf( "Back from parallel section...\n" );

        team.waitForDone();

        printf( "About to Pexit()...\n" );

        Pexit( 0 );

        return 0;
} // main



void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp )
```

```
{
        int     i, j, k;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */

        /* Process command line */

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n---------------------------------------\n" );
        printf( "2D Diffusion Using Aurora\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- How large of matrix/grid to simulate ? */

        XmatrixSize = atoi( argv[ 2 ] );  /* Get from input line: arg 2 */
        printf( "\tSquare matrix size is:  %d x %d (1 row = %d bytes)\n",
                (int)XmatrixSize, (int)XmatrixSize,
                ( (int)XmatrixSize * sizeof( float ) )
              );

        /* Set up indexing object */

        idx.set( (int)XmatrixSize, (int)XmatrixSize );

        /* Allocate space for the data */

        MatrixA.constructNow( idx.size(), (int)XnumPE );
        MatrixB.constructNow( idx.size(), (int)XnumPE );

} // setUp_NodeZeroOnly




void * doRuns( void * arg )
{
        ProcessTeam * myTeam;
        myTeam = AuroraInitThread( arg );

        int     i, j, k;
        int     m, n, p;                /* Matrix dimensions */
        int     start, end;


        m = n = p = (int)XmatrixSize;
        idx.set( (int)XmatrixSize, (int)XmatrixSize );

N_BARRIER( "Start All Runs" );          /* Wait for all PEs to start-up */

        /* Do NUM_RUNS number of test, and average statistics */

        for( RunNum = 0; RunNum < NUM_RUNS; )
        {
                /* Master will increment RunNum later */
N_BARRIER( "Start Create Data" );

MASTER
{
                /* Set the initial mass concentration (vertical bar) */

                MidPointRowCol = (int)XmatrixSize / 2;

                /* Will create a vertical band of data */
```

```
                StartMassRow = MaxIter + 1;
                EndMassRow   = (int)XmatrixSize - ( MaxIter + 1 );
                TotalMass    = ( EndMassRow - StartMassRow ) * InitVal;

                /* Deposits a unit of "InitVal" on the each portion
                        the vertical bar */

                /* Here we sprinkle mass along the vertical band */
                for( int i = StartMassRow; i < EndMassRow; i++ )
                {
                        /* NOTE:  "ii" is for block padded block */
                        MatrixA[ idx.ii( i /* row */, MidPointRowCol ) ] =
                                        InitVal;
                }
} // MASTER
END_MASTER;
N_BARRIER( "End Create Data" );

                /*
                        ----- PARALLEL -----
                */
                if( (int)XnumPE > 1 )
                {
N_BARRIER( "Simulating in PARALLEL!.  Start Timing" );

MASTER{ TotalTimer.setStart(); }
END_MASTER;


                        AuroraDiffusion2D( myTeam, MatrixA, MatrixB );

MASTER{ TotalTimer.setEnd(); }
END_MASTER;
N_BARRIER( "End Timing" );
                }
BARRIER;

MASTER
{
                /* Done simulating.  Calc statistics. */
                printf( "Done!  Run Number %d.\n",
                        (int)RunNum );

                TotalTimer.print( "\n" );
                if( (int)RunNum > 0 )
                        TotalTimer.addSample();

} // MASTER
END_MASTER;
BARRIER;

MASTER
{
                RunNum++;
} // MASTER
END_MASTER;

                fflush( stdout );
N_BARRIER( "End RunNum" );
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );


MASTER
{
```

```
        printf( "\n" );
        printf( "%d PEs / %d x %d / TimeSteps|Iterations %d / "
                "Aurora Version of 2-D Diffusion\n",
                (int)XnumPE, (int)XmatrixSize, (int)XmatrixSize,
                MaxIter );

        TotalTimer.stats( "Avg time (FINAL)\n" );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        myTeam->IamDone();

        return NULL;
} /* doRuns */




// Sequential
void Diffusion2D( float * mA, float * mB )
{
        int     i, j, k;
        int size = (int)XmatrixSize;

        /* For each iteration */
        for( int iter = 0; iter < ( MaxIter / 2 ); iter++ )
        {
                blockStencil( 0, size, 1, size, mA,  mB, idx );


                blockStencil( 0, size, 1, size, mB,  mA, idx );
        } // iteration
}
```

B.3 Parallel Sorting by Regular Sampling

B.3.1 psrs.H

```
#ifndef _PSRS_H_
#define _PSRS_H_

/*
======================================================================

        Module:         Header for PSRS

        Parallel Sorting by Regular Sampling in Aurora

======================================================================
*/

/*

        Globals

*/
#define FALSE   0
#define TRUE    1

#define PSRS_PE         16              /* Maximum number of processors */
#define PSRS_MATRIX ( PSRS_PE * ( PSRS_PE + 1 ) )

extern GScalar<int>     XnumPE;         /* Globals to pass info to all PEs */
extern GVector<int>     Xarray;
extern GScalar<int>     XarraySize;

extern int Xsize;

extern int NumPE, NumPE1;               /* Number of Processors and PE + 1 */

extern GVector<int> Pivots;             /* Selected pivots */
extern GVector<int> PSRSPartition;      /* Indices of partitions */
extern GVector<int> Bucksize;           /* Size of final buckets */
extern GVector<int> Sample;             /* Samples of locally sorted sublists */

extern int NumSample;                   /* Number of samples per PE */
extern int OverSize;
extern int Size;                        /* Size of local sublists */
extern int RSize;                       /* Regular sample distance */
extern int SampleOffset;                /* Offset; various sampling methods */


#define NUM_RUNS        6               /* Number of runs in each timing */

extern struct timeval RealTime;         /* Timestamps */
extern struct timeval tp1, tp2, tp3, tp4;
extern struct timeval Phase1Start, Phase1End;
extern struct timeval Phase2Start, Phase2End;
extern struct timeval Phase3Start, Phase3End;
extern struct timeval Phase4Start, Phase4End;

extern GScalar<int>     RunNum;         /* Current run number */

extern double Rdfa[ NUM_RUNS ];         /* Save RDFA's times for averaging */

extern GScalar<int>     Debug;          /* Flag */



/*
```

```

        Forwards

*/
extern void initPSRS( int numPE, int arraySize );
extern void PSRS( SpmdLVector<int> * myTeam,
        GVector<int> dataArray, int arraySize );
                                /* Main sorting function */

extern void partSublists( int * theArray, int * thePivots, int offset,
                int at, int start, int end, int fp, int lp );
                                /* With pivots, find partition indices*/

extern void quicksort( int * qArray, int from, int to );
                                /* Normal Quicksort with enhancements */

extern int * mergesort( int * theArray, int bid, int start, int bsize,
        int * thisPSRSPartition /* will come from a read cache */ );
                                /* Successive 2-way mergesort */

extern int * merge( int * from1, int * to1, int * from2, int * to2, int * at );
                                /* 2-way mergesort */
#endif // _PSRS_H_
```

B.3.2 main.C

```
/*
======================================================================

        Module:         main

        Parallel Sorting by Regular Sampling in Aurora

======================================================================
*/
#include "aurora.H"

#include "GScalar.H"
#include "GVector.H"

#include "GVReadCache.H"
#include "GVOwnerComputes.H"

#include "team.H"               /* For Team */
#include "oracle.H"

#include <stdio.h>
#include <stdlib.h>             /* For atof() */
#include <sys/time.h>           /* For gettimeofday(), etc. */
#include "psrs.H"

#include "oracle.H"             /* For global Aurora variables */
extern Oracle AuroraOracle;
Oracle AuroraOracle;

/*

        Globals

*/
GScalar<int>    XnumPE;         /* Globals to pass info to all PEs */

GVector<int>    Xarray;
GScalar<int>    XarraySize;

```

```
int     Xsize;

shared struct   timeval RealTime;       /* Timestamps */
shared struct   timeval tp1, tp2, tp3, tp4;
shared struct   timeval Phase1Start, Phase1End;
shared struct   timeval Phase2Start, Phase2End;
shared struct   timeval Phase3Start, Phase3End;
shared struct   timeval Phase4Start, Phase4End;

char    * DataSource;

/* Note:  First run not used in averaging */

int Seed[ 11 ] = { 2,   2, 3, 4, 5, 6,   2, 3, 4, 5, 6 };


GScalar<int>    RunNum;         /* Current run number */

double RunTimes[ NUM_RUNS ];    /* Save run times for averaging */
double Rdfa[ NUM_RUNS ];        /* Save RDFA's times for averaging */
double Phase1Times[ NUM_RUNS ]; /* Save phase run times for averaging */
double Phase2Times[ NUM_RUNS ]; /* Save phase run times for averaging */
double Phase3Times[ NUM_RUNS ]; /* Save phase run times for averaging */
double Phase4Times[ NUM_RUNS ]; /* Save phase run times for averaging */

GScalar<int> Debug;             /* Flag:  Debug */

GScalar<int> UsingRealData;     /* Flag:  Random vs real data */

int     * RealData = NULL;      /* Store input from data file */
int     RealDataSize = 0;       /* Number of data elements from file */

/* NOTE:  Only Node 0 calls psrsSetUp_NodeZeroOnly() */
extern void psrsSetUp_NodeZeroOnly( int argc, char ** argv, char ** envp );
extern void * psrsmain( void * arg );


/*
------------------------------

NAME:   Main

------------------------------
*/

int main( int argc, char ** argv, char ** envp )
{
        // NOTE:  Everybody calls Pinit(), but only Node 0 returns from it
        Pinit( argc, argv );

        // NOTE:  Only Node 0 ever calls AuroraInit()
        AuroraInit( &argc, &argv );

        // NOTE:  Only Node 0 calls psrsSetUp_NodeZeroOnly
        psrsSetUp_NodeZeroOnly( argc, argv, envp );

        // Create the team for owner-computing on Xarray

        Team team( Xarray.nodes() );

        team.parallel( psrsmain, NULL );

        printf( "Back from parallel section...\n" );

        team.waitForDone();

        printf( "About to Pexit()...\n" );
```

```
        Pexit( 0 );

        return 0;
} // main



void psrsSetUp_NodeZeroOnly( int argc, char ** argv, char ** envp )
{
        int     i, j, k;
        int     num;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */

        // Process command line

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n---------------------------------------\n" );
        printf( "Parallel Sorting by Random Sampling Using Aurora\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- How large of an array to sort ? */

        XarraySize = atoi( argv[ 2 ] ); /* Get from input line: arg 2 */
        printf( "\tSort size is:  %d\n\n", (int)XarraySize );

        /* Allocate space for the data to be sorted */

        Xarray.constructNow( (int)XarraySize, (int)XnumPE );

        /* Optional Arguments */

        UsingRealData = FALSE;
        NumSample = XnumPE;             /* Default:  No oversampling */
        Debug = FALSE;                  /* Default:  No debugging data */

        for( currentArg = 3; currentArg < argc; currentArg++ )
        {
                if( currentArg >= argc )
                        break;
                cp = argv[ currentArg ];
                if( *cp == '-' ) {
                        cp++;           /* Next field is the argument */
                        if( *cp == 's' ) {
                                NumSample = atoi( argv[ ++currentArg ] );
                        }
                        else if( *cp == 'o' ) {
                                NumSample = ( int )( ( double ) XnumPE *
                                        atof( argv[ ++currentArg ] ) );
                        }
                        else if( *cp == 'd' ) {
                                Debug = TRUE;
                        }
                        else {
                            printf( "Error:  invalid '-%c' option.  Ignored.\n",
                                        *cp );
                        }
                }
                else {
                        printf( "Error:  invalid parameter.  Ignored.\n" );
                }
        } /* for */
```

```

	/* Allocate storage for Regular Samples */

	Sample.constructNow( (int)NumSample * (int)XnumPE, Nodes( 0 ) );

	printf( "\tSize of regular sample:  %d (%d entries)  ",
			NumSample, NumSample * XnumPE );
	if( (int)NumSample > (int)XnumPE ) {
		printf( "OVERSAMPLING at %f",
			(float)NumSample / (float)XnumPE );
	}
	printf( "\n" );
	fflush( stdout );

} /* psrsSetUp_NodeZeroOnly */



void * psrsmain( void * arg )
{
	ProcessTeam * myTeam;
	myTeam = AuroraInitThread( arg );

	int	i, j, k;
	int	start, end;
	int	sec, usec;
	int	sumSec, sumUsec;
	double	timeSum;
	double	rdfaSum;
	double	phase1Sum;
	double	phase2Sum;
	double	phase3Sum;
	double	phase4Sum;
	int	numErrors;

	int	ii, kk, num;
	int	currentArg;	/* Current command line argument */
	char	* cp;		/* For reading line arguments */
	float	tr, t1;		/* For calculating statistics */

BARRIER;			/* Wait for all PEs to start-up */

	/* Do NUM_RUNS number of test, and average statistics */

	for( RunNum = 0; RunNum < NUM_RUNS; )
	{
		/* Master will increment RunNum later */

MASTER
{
		printf( "\n\nRun Number %d (warm-up %d), Seed is %d\n",
			(int)RunNum, Seed[ RunNum ] );
		printf( "Sorting %d numbers on %d PEs with seed %d..\n",
			(int)XarraySize, (int)XnumPE, Seed[ RunNum ] );
		printf( "Creating Random data in parallel...\n" );
} // MASTER
END_MASTER;
BARRIER;

		Xsize = ( XarraySize + XnumPE - 1 ) / XnumPE;

BARRIER;		/* Wait for XSize, XarraySize */

		if( ! UsingRealData )
		{

			/* Create data in parallel */
```

```

                    { // New scope
                        NewBehaviour( Xarray, GVOwnerComputes, int );

                        while( Xarray.doParallel( myTeam ) )
                        {
                                i = myTeam->getId();

                                start = Xarray.begin();
                                end   = Xarray.end() - 1;

                                /* Fill [ start : end ] with data */

                                srand( i * Seed[ RunNum ] );
                                for( k = start; k <= end; k++ )
                                {
                                        Xarray[ k ] = rand();  /* Create data */
                                }
                        } // doParallel
                    } // End scope
                } // if
N_BARRIER( "End Data Creation" );       // Wait for all data to be created

                /*
                        ----- PARALLEL -----
                */
                initPSRS( XnumPE, XarraySize );  /* Yes, use PSRS */

                PSRS( myTeam, Xarray, (int)XarraySize );
BARRIER;

MASTER
{
                /* Done sorting.  Calc statistics. */
                printf( "Done sorting!  Run Number %d.\n", (int)RunNum );
} // MASTER
END_MASTER;
N_BARRIER( "End sorting" );             /* Wait until all PEs done */

            { // New scope
                NewBehaviour( Xarray, GVOwnerComputes, int );

                while( Xarray.doParallel( myTeam ) )
                {
                        i = myTeam->getId();

                        start = Xarray.begin();
                        end   = Xarray.end() - 2; // Since checking j and j + 1

                        numErrors = 0;
                        for( j = start; j <= end; j++ )
                        {
                                if( Xarray[ j ] > Xarray[ j + 1 ] &&
                                        ( numErrors < 10 ) )
                                {
                                        printf( "*Order error (%d): ", _IPROC );
                                        printf( "j %d = %d, j+1 %d = %d\n",
                                                j, Xarray[ j ], j + 1,
                                                Xarray[ j + 1 ] );
                                }
                        }

                        if( numErrors > 0 )
                        {
                                printf( "-- at least %d order errors in %d!\n",
                                        numErrors, i );
                        }
```

```
                } // while/doParallel
            } // End scope
N_BARRIER( "End Verify (in main)" );

MASTER
{
                RunNum++;               /* Ready to do next run */
} // MASTER
END_MASTER;
N_BARRIER( "End RunNum" );

                fflush( stdout );
BARRIER;
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );


MASTER
{
        printf(
    "\n%d PEs (%d) / %d / Off %d / Sample %d / Avg time %d runs: %f (%s) / "
    "Aurora Version of PSRS\n",
                        (int)XnumPE, _NPROCS, (int)XarraySize, SampleOffset,
                        NumSample, NUM_RUNS - 1,
                tr = (float)( ( timeSum / ( NUM_RUNS - 1 ) ) / 1000000 ),
                        DataSource );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        myTeam->IamDone();

        return NULL;
} /* psrsmain */
```

B.3.3 psrs.C

```
/*
======================================================================

        Module:         PSRS

        Parallel Sorting by Regular Sampling in Aurora

======================================================================
*/
#include "aurora.H"

#include "GScalar.H"
#include "GVector.H"

#include "GVReadCache.H"
#include "GVReadCacheSparse.H"  /* Allow caching, but do not load eagerly */
#include "GVReleaseC.H"
#include "GVOwnerComputes.H"

#include "distmemcpy.H"

#include "team.H"               /* For Team */

#include <stdio.h>
```

```
#include <sys/time.h>           /* For gettimeofday(), etc. */
#include <string.h>             /* For memcpy() */
#include "psrs.H"

#include "oracle.H"             /* Need if I have Aurora global variables */
extern Oracle AuroraOracle;

/*

        Globals

*/
int NumPE, NumPE1;              /* Number of Processors and PE + 1 */

int MergeBufferSize;            /* Buffers used in mergesort */

GVector<int> Sample             /* Samples of locally sorted sublists */

/* We do owners-computes in a MASTER{} later, so we need this on Node 0 */
GVector<int> Pivots( PSRS_PE, Nodes( 0 ) );     /* Selected pivots */

/* Co-located with master */
GVector<int> PSRSPartition( PSRS_MATRIX, Nodes( 0 ) );
                                /* Indices of partitions */
GVector<int> Bucksize( PSRS_PE, Nodes( 0 ) );
                                /* Size of final buckets */

int NumSample;                  /* Number of samples per PE */
int OverSize;
int Size;                       /* Size of local sublists */
int RSize;                      /* Regular sample distance */

int SampleOffset;               /* Offset; various sampling methods */

static bool MergeBuffersInitialized = false;    /* Flag:  only memset once */
static int * sortedListArray[ PSRS_PE ];
static int * holdBufferArray[ PSRS_PE ];

TimeObj TotalTime( "Total Sort Time (all phases):" );

/*
------------------------------

NAME:   initPSRS

------------------------------

        Initialize the global variables required before calling PSRS.
*/
void initPSRS( int numPE, int arraySize )
{
        // Set a bunch of global variables that are precomputed to save
        //      on effort later.  Only one process per-address space
        //      needs to compute this, but there is no real harm
        //      in multiple processes doing it.

        NumPE = numPE;
        NumPE1 = NumPE + 1;
        NumSample = NumPE;              /* No over/under-sampling */

        Size = ( arraySize + NumPE - 1 ) / NumPE; /* Size of local sub-list */
        RSize = ( Size + NumPE - 1) / NumPE;      /* Regular sample interval */
        OverSize = ( Size + NumSample - 1 ) / NumSample;
                                                /* Oversample interval */
        // Only do once.
        // Per-thread buffers for mergesort.  Initialize all to un-allocated.
        if( MergeBuffersInitialized == false )
```

```
        {
                memset( sortedListArray, NULL, sizeof( sortedListArray ) );
                memset( holdBufferArray, NULL, sizeof( holdBufferArray ) );

                // Size of buffers (for sorting and holding) for mergesort
                // Estimate maximum need based on problem size and num pe's
                MergeBufferSize = (int)( ( arraySize / numPE ) * 1.2 );

                MergeBuffersInitialized = true;
        }
} /* initPSRS */






/*
------------------------------

NAME:   PSRS

------------------------------

        Parallel Sorting by Regular Sampling.  You must call
        initPSRS() once before you use this function,
        and whenever you want to change the number of processors
        to be used.
*/
void PSRS( ProcessTeam * myTeam,
           GVector<int> dataArray, int arraySize )

//      myTeam    --    the SPMD team to execute this function
//      dataArray --    Location of data array of ints
//      arraySize --    Number of integers in array
{
        int     i;              // Process ID.  Do NOT use for loop index.
        int     start, end;     /* [ start : end ] is a local sublist */
        int     maxBucksize;    /* Maximum partition size.  For RDFA */
        int     ii, j, k;       /* Indices */

        /*

                Phase 1 -- Sort local sublists and collect Regular Sample

        */

        /* Equal to:  for i = 0 to p - 1 do in parallel */

BARRIER;                        /* Wait for Size and RSize */

N_BARRIER( "Start PSRS" );

MASTER
{
        TotalTime.setStart();

        gettimeofday( &RealTime, 0 );           /* Real time */
        gettimeofday( &tp1, 0 );                /* Phase 1 */
        gettimeofday( &Phase1Start, 0 );        /* New */
} // MASTER
END_MASTER;

    { // New scope
        NewBehaviour( dataArray, GVOwnerComputes, int );
        dataArray.doParallel( myTeam );

```

```
                i       = myTeam->getId();
                start   = dataArray.begin();
                end     = dataArray.end() - 1;

                /* Sequentially Quicksort local sublist */

                /* Sorts from [ 0, end - start ] *inclusive* */
                quicksort( &( dataArray[ start ] ), 0, end - start );

    } // End scope
N_BARRIER( "End Local Sort" );  /* Wait for all sublists to be sorted */

    { // New Scope
        NewBehaviour( dataArray, GVOwnerComputes, int );
        NewBehaviour( Sample, GVReleaseC, int );
        dataArray.doParallel( myTeam );

                i       = myTeam->getId();
                start   = dataArray.begin();
                end     = dataArray.end() - 1;

                /* Sample each local sublist into Sample Matrix */

                for( j = 0; j < NumSample; j++ )
                {
                        int sampleIndex = ( i * NumSample ) + j;
                        int dataIndex = start + ( j * OverSize );
                        if( dataIndex > end )
                                dataIndex = end;

                        /* Copy the data for the sample */
                        Sample[ sampleIndex ] = dataArray[ dataIndex ];
                } // for
    } // End scope

N_BARRIER( "End Local Sample" );

MASTER { gettimeofday( &Phase1End, 0 ); } END_MASTER;

        /*

                Phase 2 -- Sort the Regular Sample and choose the pivots

                           !!!(Done by the Master process only !!!

        */

MASTER
{
        printf( "Individual sublists sorted.\n" );
        gettimeofday( &tp3, 0 );
        gettimeofday( &Phase2Start, 0 );

    { // New Scope
        NewBehaviour( Sample, GVOwnerComputes, int );   // NOTE: Inside MASTER{}
        NewBehaviour( Pivots, GVOwnerComputes, int );   // NOTE: Inside MASTER{}

        Sample.doParallel( myTeam );
        Pivots.doParallel( myTeam );

        /* Sort Regular Sample */

        /* Sorts from [ 0, NumPE * NumSample - 1 ] *inclusive* */
        quicksort( &( Sample[ 0 ] ), 0, NumPE * NumSample - 1 );

        /* Select pivots */

```

```
        k = NumSample / 2 - 1;          /* Location of initial pivot */
        if( k < 0 )
        {
                k = 0;
        }
        SampleOffset = k;               /* Remember sampling offset */

        // NOTE:  Need to start loop at zero.  The formula for k above
        //      picks poor pivots if we don't start at zero.
        for( ii = 0; ii < NumPE; ii++ )
        {
                Pivots[ ii ] = Sample[ k ];
                k += NumSample;
        } // for

    } // End scope

} // MASTER
END_MASTER;
N_BARRIER( "End Pivot Selection" );             /* Wait for the pivots */

MASTER
{
        gettimeofday( &tp4, 0 );
        gettimeofday( &Phase2End, 0 );
}
END_MASTER;

        /*

                Phase 3 -- Create/Find and Exchange partitions
                        (May not be an exchange for shared memory)

        */

MASTER { gettimeofday( &Phase3Start, 0 ); } END_MASTER;

        /* Decide on location of sublists and bucket size */

    { // New scope
        NewBehaviour( dataArray, GVOwnerComputes, int );
        NewBehaviour( Pivots, GVReadCache, int );  // NOT in MASTER, must cache
        NewBehaviour( PSRSPartition, GVReleaseC, int );

        while( dataArray.doParallel( myTeam ) )
        {
                i       = myTeam->getId();
                start   = dataArray.begin();
                end     = dataArray.end() - 1;

                PSRSPartition[ i * NumPE1 ] = start;
                PSRSPartition[ i * NumPE1 + NumPE ] = end + 1;

                partSublists( &( dataArray[ start ] ), Pivots, start,
                        i * NumPE1, start, end, 1, NumPE - 1 );
        } // doParallel
    } // End scope
N_BARRIER( "End Compute Partitions" );  /* Wait for partitions */

        /* Calculate the size of the different buckets */

MASTER
{
    { // New scope
        NewBehaviour( Bucksize, GVOwnerComputes, int );      // For read-write
        NewBehaviour( PSRSPartition, GVOwnerComputes, int ); // For read

```

```
        Bucksize.doParallel( myTeam );
        PSRSPartition.doParallel( myTeam );

        for( ii = 0; ii < NumPE; ii++ )
        {
                Bucksize[ ii ] = 0;

                for( k = ii; k < ( NumPE * NumPE1 ); k += NumPE1 )
                {
                        int rhs = PSRSPartition[ k + 1 ] - PSRSPartition[ k ];
                        int lhs = Bucksize[ ii ];
                        Bucksize[ ii ] = lhs + rhs;
                }
        }

        // Find the larget bucket

        maxBucksize = Bucksize[ 0 ];
        for( ii = 1; ii < NumPE; ii++)
        {
                if( Bucksize[ ii ] > maxBucksize ) {
                        maxBucksize = Bucksize[ ii ];
                }
                int rhs = Bucksize[ ii - 1 ];
                int lhs = Bucksize[ ii ];
                Bucksize[ ii ] = lhs + rhs;
        }

        Rdfa[ RunNum ] = (float)( (float)maxBucksize /
                        ( (float)arraySize / (float)NumPE ) );
        printf( "Largest partition is %d.  Perfect is %d.  RDFA = %f\n",
                        maxBucksize, arraySize / NumPE,
                        (double)Rdfa[ RunNum ] );
    } // End scope
} // MASTER
END_MASTER;
N_BARRIER( "End Compute Bucket Sizes" );        /* Wait for bucket sizes */

        int * myMergedBucket = NULL;
        int myBucketSize = 0;
        int myBucketStart = 0 ;

    { // New scope (very large)

        NewBehaviour( dataArray, GVReadCacheSparse, int );// Partial read cache
        NewBehaviour( PSRSPartition, GVReadCache, int ); // NOT in MASTER, must cache

        i = myTeam->getId();

        int bytesWanted = 0;
        int bytesGrabbedTotal = 0;
        int bytesGrabbedThisTime = 0;

        int j = i + ( ( i + 1 ) * NumPE1 );     // Space out the starting points
        for( int dinloop = 0; dinloop < NumPE; dinloop++ )
        {
                // Wrap-around the "real" loop index, if need
                if( j >= ( NumPE * NumPE1 ) )
                        j -= ( NumPE * NumPE1 );

                int start = PSRSPartition[ j ];
                int stop  = PSRSPartition[ j + 1 ];

                bytesWanted += distmemcpy( &( ( (int*)dataArray)[ start ] ),
                                  (*dataArray._saveHandle)[ start ],
                                  ( stop - start ),
                                & bytesGrabbedThisTime
```

```
                        );

                bytesGrabbedTotal += bytesGrabbedThisTime;

                // Advance, with wrap-around (above)
                j += NumPE1;
        } // for-loop

// Careful, inside a scope here
N_BARRIER( "End Get Partitions" );

    { // New scope
        NewBehaviour( Bucksize, GVReadCache, int );       // Full read cache
        if( i )
        {
                myBucketStart = (int)Bucksize[ i - 1 ];
                myBucketSize  = (int)Bucksize[ i ] - (int)Bucksize[ i - 1 ];
        }
        else
        {
                myBucketStart = 0;
                myBucketSize = (int)Bucksize[ 0 ];
        }
    } // End scope

// Careful, inside a scope here
N_BARRIER( "End Get Partitions and Buckets" );

        /*

                Phase 4 -- Final merge

        */

MASTER
{
        gettimeofday( &Phase3End, 0 );
        gettimeofday( &Phase4Start, 0 );
} // MASTER
END_MASTER;

MASTER { printf( "Final merging method is SUCCESSIVE...\n" ); } END_MASTER;

        // PSRSPartition was read-cached above
        myMergedBucket = mergesort(
                dataArray, i, myBucketStart, myBucketSize,
                PSRSPartition /* now passing as int * since I have a cache */
                );
    } // End scope (very large)

        /* Copy back results to the original Array */

        // Copy merged data back to shared vector
        distmemcpy( dataArray[ myBucketStart ], myMergedBucket, myBucketSize );

N_BARRIER( "End PSRS" );                /* Wait until all done */

MASTER
{
        gettimeofday( &tp2, 0 );
        gettimeofday( &Phase4End, 0 );          /* New */

        TotalTime.setEnd();

} // MASTER
END_MASTER;
```

```
        /*

                Sorting Over.  Verify results.

        */

MASTER { printf( "Verifying order of data...\n" ); } END_MASTER;

    { // New scope
        NewBehaviour( dataArray, GVOwnerComputes, int );

        while( dataArray.doParallel( myTeam ) )
        {
                i       = myTeam->getId();
                start   = dataArray.begin();
                end     = dataArray.end() - 1;

                // Check ordering *within* a local block

                numErrors = 0;
                for( j = start; j < end; j++ )
                {
                        if( dataArray[ j ] > dataArray[ j + 1 ] ) {
                                printf( "PSRS:  Order error (%d): ", _IPROC );
                                printf( "j %d = %d, j+1 %d = %d\n",
                                        j, dataArray[ j ], j + 1,
                                        dataArray[ j + 1 ] );
                        }
                }
        } // doParallel
    } // End scope
N_BARRIER( "End Verify (in psrs)" );

        // Check boundary conditions:  *spanning* local blocks

MASTER { printf( "Verifying order of boundary data...\n" ); } END_MASTER;

        if( end < ( arraySize - 1 ) )
        {
                if( (int)dataArray[ end ] > (int)dataArray[ end + 1 ] )
                {
                        printf( "PSRS:  Boundary Order error (%d): ", _IPROC );
                        printf( "j %d = %d, j+1 %d = %d\n",
                                end, (int)dataArray[ end ], end + 1,
                                (int)dataArray[ end + 1 ] );
                        numErrors++;
                }
        }
N_BARRIER( "End Boundary Verify" );
} /* PSRS */







/*
------------------------------

NAME:   partSublists

------------------------------

        Compute location of partitions in each local sublist,
        given a set of pivots.  Done via a binary search.
*/
```

```

void partSublists( int * theArray, int * thePivots, int offset,
                int at, int start, int end, int fp, int lp )
{

        register int mid, center, lb, ub, pv;

        mid = ( fp + lp ) / 2;
        pv = thePivots[ mid ];

        lb = start;
        ub = end;
        while( lb <= ub ) {
                center = ( lb + ub ) / 2;
                if( theArray[ center - offset ] > pv ) {
                        ub = center - 1;
                }
                else {
                        lb = center + 1;
                }
        }

        PSRSPartition[ at + mid ] = lb;

        if( fp < mid ) {
                partSublists( theArray, thePivots, offset,
                        at, start, lb - 1, fp, mid - 1 );
        }
        if( mid < lp ) {
                partSublists( theArray, thePivots, offset,
                        at, lb, end, mid + 1, lp );
        }
} /* partSublists */





/*
------------------------------

NAME:   quicksort

------------------------------

        Quicksorts the interval Array[from:to]
        from    left end of interval
        to      right end of interval

*/
        // Sorts from [ from, to ] *inclusive*
void quicksort( int * qArray, int from, int to )
{
        // Purely sequential code.  Omitted.
} // quicksort




/*
------------------------------

NAME:   mergesort

------------------------------

        merge pe sublists of bucket bid
```

```
        bid     id of the bucket
        start   from where the merged data are to put
        bsize   Size of the bucket
*/
int * mergesort( int * theArray, int bid, int start, int bsize,
        int * thisPSRSPartition /* will come from a read cache */
)
{
        // Purely sequential code.  Omitted.
} /* mergesort */






/*
------------------------------

NAME:   merge

------------------------------

        Straightforward 2-way merge of specified lists
*/

int * merge( int * from1, int * to1, int * from2, int * to2, int * at )
{
        // Purely sequential code.  Omitted.
} /* merge */
```

B.4 Travelling Salesperson Problem

B.4.1 tsp.h

```
#define BROAD_TOUR

        /* Paul: 32 comes from 32-bits of field 'conn' in TourElement  */
#define MAX_TOUR_SIZE   32
#define MAX_NUM_TOURS   10000
#define BIGINT          2000000
#define END_TOUR        (-1)
#define ALL_DONE        (-1)

#define BEEP            7

#include <memory.h>

class TourElement
{
public:
    int prefix[MAX_TOUR_SIZE];  /* List of nodes in cur tour. */
        /* Paul: i.e. use the bit-map of 'conn'  */
    int conn;                   /* conn & 1<<i => in cur tour. */
    int last;                   /* Index of last node in cur tour. */
    int prefix_weight;          /* Weight of edges in cur tour. */
    int lower_bound;            /* Lower bound of edges not in cur tour. */
    int mst_weight;             /* minimum spanning tree cost for rest. */

        // Ctor
        TourElement() {}        // Does nothing, for now

        TourElement & operator=( const TourElement & rhs )
        {
                // Only proceed if not the same object (i.e., not "a = a")
                if( this != & rhs )
                {
                        memcpy( this, &rhs, sizeof( TourElement ) );
                }
                else printf( "Error in class TourElement\n" );

                return *this;   // By convention...
        } // operator= (assignment)
}; // TourElement

class PrioQElement
{
public:
    int  index;                 /* Index into tours[] of this element. */
    int  priority;              /* Priority of this element. */

        // Ctor
        PrioQElement() : index( 0 ), priority( 0 ) {}

        PrioQElement & operator=( const PrioQElement & rhs )
        {
                // Only proceed if not the same object (i.e., not "a = a")
                if( this != & rhs )
                {
                        // Do a (slower?) field-by-field copy for now
                        this->index     = rhs.index;
                        this->priority  = rhs.priority;
                }
                else printf( "Error in class PrioQElement\n" );

                return *this;   // By convention...
        } // operator= (assignment)
```

```

}; // PrioQElement

extern weights[MAX_TOUR_SIZE][MAX_TOUR_SIZE];

extern int read_tsp( char * file);
extern void set_best( int best, int * path);


extern int calc_bound(int curr_index, TourElement * currTour );
extern int new_tour(int prev_index, int move, TourElement * prevTour,
                        TourElement * newTour );
extern void recursive_solve(int index, TourElement * theTour );

extern void MakeTourString(int len, int * path);

extern void PrintTourString( char * annotation );

extern char *fgets();

extern int TspSize, StartNode;
extern char tour_str[];

extern int NodesFromEnd, debug, debugPrioQ;
extern int DDThreshold, DDSaveDebug, DDSaveDebugPrioQ;
extern int visitNodes;

extern void DumpPrioQ( char * annot, int lowerBound = 0 );

extern void print_comments();
extern int theExpectedAnswer();
```

B.4.2 tsp.H (for Aurora)

```
#ifndef _TSP_H_
#define _TSP_H_

/*
======================================================================

        Travelling Salesperson Problem

        Ported to Aurora.  Originally from Rice group.

======================================================================
*/

#include "aurora.H"

#include "GScalar.H"
#include "GVector.H"

#include "GVOwnerComputes.H"            // Scoped behaviours
#include "GVReadCache.H"
#include "GVReleaseC.H"
#include "GSOwnerComputes.H"            // Scalar

#include "team.H"                       // For Team
#include "oracle.H"

#define NUM_RUNS        6               // Number of runs in each timing

/*

        Forwards
```

```

*/
extern GScalar<int> XnumPE;
extern GScalar<int> RunNum;
extern GScalar<int> Debug;

extern TimeObj  ParTimer;
extern TimeObj  SeqTimer;

extern GScalar<int>             GS_Done;
extern GScalar<int>             GS_MinTourLen;
extern GVector<int>             GV_MinTour;
extern GScalar<int>             GS_PrioQLast;
extern GVector<PrioQElement>    GV_PrioQ;
extern GScalar<int>             GS_TourStackTop;
extern GVector<int>             GV_TourStack;
extern GVector<TourElement>     GV_Tours;

// These variables are read-only.  These are used only to once-copy
//      the data into each address space.
extern GScalar<int>             Global_TspSize;
extern GScalar<int>             Global_NodesFromEnd;
extern GScalar<int>             Global_debug;
extern GScalar<int>             Global_StartNode;
extern GScalar<int>             Global_debugPrioQ;
extern GScalar<int>             Global_DDThreshold;
extern GVector<int>             Global_weights;

extern int get_tour(int curr, TourElement * currTour );

class TourPackage
{
    public:
        TourElement     theTour;
        int             tourIndex;
}; // TourPackage

class GetTourServer : public MyABC            // Active object.  ABC++
{
    private:
    public:
        // Constructor
        GetTourServer() { }

        // _range not set
        GetTourServer(int id) : MyABC( id )
        {
                printf( "%d : GetTourServer is running\n", MYPROC );
        } // Ctor

        // Dtor
        ~GetTourServer()        { }

        TourPackage remote_get_tour_package( int curr )
        {
                TourPackage tempTP;
                tempTP.tourIndex = get_tour( curr, &( tempTP.theTour ) );
                return( tempTP );
        } // remote_get_tour_package

}; // GetTourServer

extern Pabc_pointer<GetTourServer> TheGetTourServer;

extern ProcessTeam * GlobalMyTeam;

extern TourPackage CacheTour;
```

```

#endif // _TSP_H_
```

B.4.3 main.C

```
/*
======================================================================

        Travelling Salesperson Problem

        Ported to Aurora.

======================================================================
*/

#include <stdio.h>
#include <memory.h>             /* For memcpy() */
#include <unistd.h>             /* For getopt() */

#include "tsp.h"
#include "tsp.H"                /* For Aurora's needs */

Oracle AuroraOracle;

/*

        Globals

*/
GScalar<int>    XnumPE;         /* Globals to pass info to all PEs */
char * InputFilename = "<none>";

GScalar<int> RunNum;            /* Current run number */
GScalar<int> Debug;             /* Flag:  Debug */

TimeObj ParTimer    ( "Total time in PARALLEL:" );
TimeObj SeqTimer    ( "Total SEQUENTIAL time :" );

StatsObj<int,NUM_RUNS> NodesSearchedStats( "Nodes searched: " );
StatsObj<int,NUM_RUNS> NumToursStats     ( "Tours searched: " );

double RunTimes[ NUM_RUNS ];    /* Save run times for averaging */

// NOTE:  Only Node 0 calls setUp_NodeZeroOnly()
extern void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp );
extern void * doTsp( void * arg );

#include <sys/time.h>

/* BEGIN SHARED DATA */

GScalar<int>            GS_Done( Nodes( 1 ) );
GScalar<int>            GS_MinTourLen( Nodes( 1 ) );
GVector<int>            GV_MinTour( MAX_TOUR_SIZE, Nodes( 1 ) );

        // PQ is accessed with owner computes
GScalar<int>            GS_PrioQLast( Nodes( 0 ) );
GVector<PrioQElement>   GV_PrioQ( MAX_NUM_TOURS, Nodes( 0 ) );

        // GV_TourStack keeps track of the empty entries/tours in GV_Tours[]
        //      Used with owner computes
GScalar<int>            GS_TourStackTop( Nodes( 0 ) );
GVector<int>            GV_TourStack( MAX_NUM_TOURS, Nodes( 0 ) );
GVector<TourElement>    GV_Tours( MAX_NUM_TOURS, Nodes( 0 ) );

```

```
GSemaphore<int> GS_MinLock;

/* END SHARED DATA */


int             StartNode, TspSize, NodesFromEnd;
// Variables with Global_* prefix are used to make local copies of
//      (generally) read-only data.
//      These shared-data objects will be "destroyed" after they have been used
GScalar<int>    Global_TspSize;

        /* Paul: 'weights' is write-once, read-many  */
int             weights[MAX_TOUR_SIZE][MAX_TOUR_SIZE];
GVector<int>    Global_weights( ( MAX_TOUR_SIZE * MAX_TOUR_SIZE ),
                                        Nodes( 0 ) );

GScalar<int>    Global_NodesFromEnd;
GScalar<int>    Global_StartNode;

GVector<int>    Global_visitNodes( 16 );
GVector<int>    Global_countTours( 16 );
extern int      visitNodes;     // Local to a node (i.e., per-node)

extern int get_tour(int curr, TourElement * currTour );

Pabc_pointer<GetTourServer> TheGetTourServer;

ProcessTeam * GlobalMyTeam;  // Only allowed one team per-address space then

TourPackage CacheTour;

// Forward def this here instead of tsp.h because
//      ProcessTeam is Aurora-specific
extern void Worker( ProcessTeam * myTeam );

void usage()
{
fprintf(stdout, "tsp:\n\t%s\n\t%s\n\t%s\n\t%s\n\t%s\n\t%s\n\t%s\n",
                "-n: number of hosts", "-s: start node",
                "-f: size (filename)", "-r: size to recursively solve",
                "-d: debug mode", "-D: debug the priority queue",
                "-S: run silently");
} // usage

extern char    *optarg;


// This function is executed SPMD
void tsp_orig_main_1( ProcessTeam * myTeam, int argc, char ** argv )
{
    int                 c, i, j, silent;
    char               *fname;
    struct timeval      start, end;

    TspSize = MAX_TOUR_SIZE;
    NodesFromEnd = 12;
    fname = "none";     // No real default

MASTER
{
    if (argc == 1) usage();

        // NOTE:  Everybody parses the command line arguments.
        //      Should be OK if this is done after MPI is initialized.
    for( i = 1; i < argc; i++ )
    {
        if( argv[ i ][ 0 ] != '-' )
```

```
                continue;

        c = argv[ i ][ 1 ];

        switch (c) {
        case 'f':
                fname = argv[ i + 1 ];
                i++;
                break;
        case 'r':
        /* Paul: Threshold to stop splitting the tour for more work */
        /* Paul: Default to 12 above.  Used in pq.c in particular. */
                sscanf(argv[ i + 1 ], "%d", &NodesFromEnd);
                i++;
                break;
        case 's':
        /* Paul:  Used in solve.c */
                StartNode = atoi(argv[ i + 1 ]);
                i++;
                printf("%d : Starting at node %d\n", MYPROC ,StartNode);
                break;
        case '?':
        case 'h':
                usage();
                exit(0);
        default:
                printf("%d : Unknown flag '%c' skipped.\n", MYPROC, c );
        }
    } // for
    fflush( stdout );

        // Copy variables to global scalar to share it
        Global_NodesFromEnd = NodesFromEnd;
        Global_StartNode = StartNode;

} // MASTER
END_MASTER;
N_BARRIER( "End Parse Command Line" );
fflush( stdout );

        // Copy variables to global scalar to share it
        NodesFromEnd = (int)Global_NodesFromEnd;
        StartNode = (int)Global_StartNode;

N_BARRIER( "End Copy Out Globals" );
fflush( stdout );

    MASTER
    {
        /* Read in the data file with graph description. */
        TspSize = read_tsp(fname);

        // Master copies local data into shared data.  Below, everyone
        //      (including the master) copies from shared to local data.
        Global_TspSize = TspSize;
        distmemcpy( Global_weights[ 0 ],
                (int*)weights, (int)( MAX_TOUR_SIZE * MAX_TOUR_SIZE ) );

    } // MASTER
    END_MASTER;
N_BARRIER( "End read file" );

        /* Initialize the first tour. */
        // Split copy:  Emulating Tmk_distribute().  See above.
        TspSize = Global_TspSize;
        distmemcpy( (int*)weights, Global_weights[ 0 ], Global_weights.size() );

```

```
N_BARRIER( "End propagate weights" );
fflush( stdout );

        // We are done with these now
        Global_TspSize.destroy( "Global_TspSize" );     // Get rid of this now
        Global_weights.destroy( "Global_weights" );
        Global_NodesFromEnd.destroy( "Global_NodesFromEnd" );
        Global_StartNode.destroy( "Global_StartNode" );

N_BARRIER( "End destroy unnecessary global objects" );
fflush( stdout );

        // Select a policy of replication with eager update for these scalars.
        //   Since these are globals, we use methods to set new behaviour.
        //   So the following is equivalent to GSReadMostly.
        GS_Done.setEagerUpdate();
        GS_MinTourLen.setEagerUpdate();

N_BARRIER( "End set update mode");

MASTER
{
        Pabc_create( Pproc( 0 ), TheGetTourServer, 0 );

        for( int i = 1; i < P_RTS.size(); i++ )         // Starts at ONE
        {
                // A literal/explicit copy into another node's address space
                putMemBlock( i, &TheGetTourServer,
                        &TheGetTourServer, sizeof( TheGetTourServer ) );
        }
} // MASTER
END_MASTER;
N_BARRIER( "TheGetTourServer started" );

} // tsp_orig_main_1




// This function is executed SPMD
void tsp_orig_main_2( ProcessTeam * myTeam, int argc, char ** argv )
{
    int                 i, j, silent;
    struct timeval      start, end;

        // Everybody do this
        visitNodes = 0;

MASTER
{
        // GV_TourStack keeps track of the empty entries/tours in GV_Tours[]

        // Some additional inits that need to be done if we want
        //      a run-loop of 5 runs
        GS_Done = 0;
        // GS_PrioQLast is reset below
        GV_MinTour.zeroElements();
        GV_PrioQ.zeroElements();
        GV_TourStack.zeroElements();
        GV_Tours.zeroElements();

        Global_visitNodes.zeroElements();
        Global_countTours.zeroElements();

        TourElement cache_GV_Tour_zero;
        cache_GV_Tour_zero.prefix[ 0 ] = StartNode;
        cache_GV_Tour_zero.conn = 1;
```

```
        cache_GV_Tour_zero.last = 0;
        cache_GV_Tour_zero.prefix_weight = 0;

        /* Paul:  Simply heuristic to compute lower-bound (greedy min) */
        calc_bound( 0, &cache_GV_Tour_zero );   /* Sets lower_bound. */

        GV_Tours[ 0 ] = cache_GV_Tour_zero;     /* Write back */

        PrioQElement tPrioQ_1;
        tPrioQ_1.index = 0;
        /* Paul:  lower_bound is computed and set in calc_bound() */
        tPrioQ_1.priority = cache_GV_Tour_zero.lower_bound; /* Read-only RHS */
        GV_PrioQ[ 1 ] = tPrioQ_1;

        GS_PrioQLast = 1;

        /* Put all the unused tours in the free tour stack. */
        /* GV_TourStack keeps track of the empty entries/tours in GV_Tours[] */
        int temp = GS_TourStackTop;
        for (i = MAX_NUM_TOURS - 1; i > 0; i--)
        {
                GV_TourStack[ ++temp ] = i;
        } // for
        GS_TourStackTop = temp;         /* Write back */

} // MASTER
END_MASTER;
BARRIER;

N_BARRIER( "End PQ init : Start parallel timing" );

        gettimeofday(&start, NULL);

    { TimeScope timer( ParTimer, false );

        Worker( myTeam );
        // Barrier at end of Worker()

    } // End timer

        gettimeofday(&end, NULL);

N_BARRIER( "Start Output" );
MASTER
{
    fprintf(stdout, "%d : Elapsed time: %.2f seconds\n", MYPROC,
            (((end.tv_sec * 1000000.0) + end.tv_usec) -
             ((start.tv_sec * 1000000.0) + start.tv_usec)) / 1000000.0);

    fprintf(stdout, "\n%d : -----------------\n\n", MYPROC );

        int nodesTotal = 0;
        int toursTotal = 0;
        int nTemp, tTemp;
        for( i = 0; i < RTS_NPROCS; i++ )
        {
                nTemp = Global_visitNodes[ i ];
                tTemp = Global_countTours[ i ];
                nodesTotal += nTemp;
                toursTotal += tTemp;
                printf( "  Node %d : %10d nodes, %3d tours\n",
                        i, nTemp, tTemp );
        }

    fprintf(stdout, "%d : \t[MINIMUM TOUR LENGTH: %d]  "
                    "[Input: %s, NodesFromEnd %d]\n\n",
                MYPROC, (int)GS_MinTourLen, InputFilename, NodesFromEnd );
```

```

        /* Check the answer */
        if( (int)GS_MinTourLen == theExpectedAnswer() )
        {
                printf( "%d : Answer is OK %d == %d\n",
                        MYPROC, (int)GS_MinTourLen, theExpectedAnswer() );
        }
        else
        {
                printf( "%d : Error : Answer is incorrect : %d != %d\n",
                        MYPROC, (int)GS_MinTourLen, theExpectedAnswer() );
        } // if

        /* Paul:   Should print out total nodes visited here */
    fprintf(stdout, "%d : TOTAL NODES : %10d nodes, %3d tours\n",
                MYPROC , nodesTotal, toursTotal );

        if( (int)RunNum > 0 )
        {
                NodesSearchedStats.addTheSample( nodesTotal );
                NumToursStats.addTheSample( toursTotal );
        }

    fprintf(stdout, "%d : MINIMUM TOUR:\n", MYPROC );

        int tourCost = 0;
        int currNode, nextNode;
        for (i = 0; i < TspSize; i++) {
            if (i % 10) fprintf(stdout, " - ");
            else fprintf(stdout, "\n\t");
            currNode = (int) GV_MinTour[i];
            nextNode = (int) GV_MinTour[i + 1]; /* Keep this curr/next simple */
            fprintf(stdout, "%2d", currNode );
            tourCost += weights[ currNode ][ nextNode ];
        }
    fprintf(stdout, " - %2d : Length = %d\n\n", StartNode, tourCost );
        fflush( stdout );
        if( tourCost != (int)GS_MinTourLen )
        {
                printf( "%d : Error : tour cost does not match tour\n",
                        MYPROC );
                fflush( stdout );
        }

        print_comments();

} // MASTER
END_MASTER;

N_BARRIER( "End Output" );

} // tsp_orig_main_2






void Worker( ProcessTeam * myTeam )
{
    int curr = -1;
    int countTours = 0;

    for (;;) {

        // Version of tour server that returns an index *and* the tour itself
        CacheTour = Pvalue( TheGetTourServer,
```

```
                GetTourServer::remote_get_tour_package, curr ); /* ABC++ */
        curr = CacheTour.tourIndex;             // Extract

        if (curr < 0) {
            break;
        }

        countTours++;           // Keep track of parallel work
        recursive_solve( curr, &( CacheTour.theTour ) );

    } // for (infinite)

        // Track some stats
        Global_visitNodes[ MYPROC ] = visitNodes;
        Global_countTours[ MYPROC ] = countTours;

N_BARRIER( "Wait for finish in Worker()" );
} // Worker



int GlobalArgc;
char ** GlobalArgv;



/*
------------------------------

NAME:   Main

------------------------------
*/

int main( int argc, char ** argv, char ** envp )
{
        int     i;

        // NOTE:  Everybody calls Pinit(), but only Node 0 returns from it
        Pinit( argc, argv );

        // NOTE:  Only Node 0 ever calls AuroraInit()
        AuroraInit( &argc, &argv );

        GlobalArgc = argc;
        GlobalArgv = argv;

        // NOTE:  Only Node 0 calls setUp_NodeZeroOnly()
        setUp_NodeZeroOnly( argc, argv, envp );

        // Create the team (not using owner-computes)
        Team team( RTS_NPROCS );        // One thread per processor

        team.parallel( doTsp );

        printf( "Back from parallel section...\n" );

        team.waitForDone();

        printf( "About to Pexit()...\n" );

        Pexit( 0 );

        return 0;
} // main
```

```

void setUp_NodeZeroOnly( int argc, char ** argv, char ** envp )
{
        int     i, j, k;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */

        /* Process command line */

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n----------------------------------------\n" );
        printf( "Travelling Salesperson Using Aurora\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- What is the input filename? */

        InputFilename = argv[ 3 ];              /* Get from input line: arg 2 */
        printf( "\tInput file:\t\t  %s\n", InputFilename );
        // NOTE:  InputFilename is not really used for anything.
        //      fname is used by the rest of the program.

} // setUp_NodeZeroOnly




void * doTsp( void * arg )
{
        ProcessTeam * myTeam;
        myTeam = AuroraInitThread( arg );

        GlobalMyTeam = myTeam;

N_BARRIER( "Start tsp_orig_main_1()" );

        tsp_orig_main_1( myTeam, GlobalArgc, GlobalArgv );

N_BARRIER( "Start All Runs" );

        /* Do NUM_RUNS number of test, and average statistics */

        for( RunNum = 0; RunNum < NUM_RUNS; )
        {

N_BARRIER( "Start tsp_orig_main_2()" );

        tsp_orig_main_2( myTeam, GlobalArgc, GlobalArgv );

N_BARRIER( "End tsp_orig_main_2()" );

MASTER
{
                /* Done.  Calc statistics. */
                printf( "Done!  Run Number %d.\n", (int)RunNum );

                ParTimer.print( "\n" );
                if( (int)RunNum > 0 )
                        ParTimer.addSample();

} // MASTER
END_MASTER;

BARRIER;
```

```

MASTER
{
                RunNum++;                       /* Ready to do next run */
} // MASTER
END_MASTER;

                fflush( stdout );
N_BARRIER( "End RunNum" );
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );

MASTER
{
        printf( "\n" );
        printf( "%d PEs / File:  %s / sizeof( TourElement ) = %d / "
                "Aurora Version of TSP\n\n",
                (int)XnumPE, InputFilename, sizeof( TourElement ) );

        ParTimer.stats( "Avg time (FINAL)\n" );

        NodesSearchedStats.statsInt( "Nodes Searched (FINAL)\n" );
        NumToursStats.statsInt( "Tours Searched (FINAL)\n" );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        myTeam->IamDone();

        return NULL;
} /* doTsp */
```

B.4.4 pq.C

```
/*
 * pq.c:
 *
 *      Contains the routines for inserting an item into the priority
 * queue and removing the lowest priority item from the queue.
 *
 */
#include <stdio.h>

#include "tsp.h"
#include "tsp.H"

#define LEFT_CHILD(x)   ((x)<<1)
#define RIGHT_CHILD(x)  (((x)<<1)+1)

extern int get_tour(int curr, TourElement * currTour );

#define my_less_than(x,y,z)     (((x)->priority  < (y)->priority) || \
                         ((x)->priority == (y)->priority) && \
                          (z[(x)->index].last > z[(y)->index].last))

/*
 * split_tour():
 *
 *  Break current tour into subproblems, and stick them all in the priority
 *  queue for later evaluation.
 *
```

```
 */
// Called by:  find_solvable_tour() (below)
//      curr_ind is the index of the tour to be split
//      This tour element has already been loaded into CacheTour.theTour

//      NOTE:  curr_Tour is purely read-only
void split_tour( int curr_ind, TourElement * curr_Tour )
{
    int n_ind, last_node, wt;
    register int i, pq, parent, index, priority;
    register TourElement *curr;
    register PrioQElement *cptr, *pptr;

        // Base the new tour on this one.  My caller will then free it up

        curr = curr_Tour;               // Tour already loaded.  Parameter.

    { // New scope
        NewBehaviour( GV_Tours, GVOwnerComputes, TourElement );
        GV_Tours.doParallel( GlobalMyTeam );

        NewBehaviourScalar( GS_PrioQLast, GSOwnerComputes, int );
        NewBehaviour( GV_PrioQ, GVOwnerComputes, PrioQElement );
        GV_PrioQ.doParallel( GlobalMyTeam );

        // A bit redundant with scoped behaviour, but keep for now.
        int cache_PrioQLast = GS_PrioQLast;

    /* Create a tour and add it to the priority Q for each possible
       path that can be derived from the current path by adding a single
       node while staying under the current minimum tour length. */
    if (curr->last != (TspSize - 1)) {
        int t1, t2, t3;
        register TourElement * newTour;

        /* Paul:  For each possible branch from this node, add it to list. */
        /* Paul:  This implies that the current tour is done with, since */
        /* Paul:  is has been superseded by the new tours, which are longer. */
        /* Paul:  See original comments above too. */

        /* NOTE:  curr and curr_Tour are read-only */

        last_node = curr->prefix[curr->last];
        // For all the child nodes of current last node in the tour
        for (i = 0; i < TspSize; i++) {
            /*
             * Check: 1. Not already in tour
             *        2. Edge from last entry to node in graph
             *  and   3. Weight of new partial tour is less than cur min.
             */
            wt = weights[last_node][i];
            t1 = ((curr->conn & (1<<i)) == 0);
            t2 = (wt != 0);

                // The use of replication with eager updates makes the following
                //      read of the shared scalar reasonable in performance.
            t3 = (curr->lower_bound + wt) <= (int)GS_MinTourLen;
            if (t1 && t2 && t3) {

        /* Paul:   Careful:  new_tour() changes values, but not curr */

                // NOTE:  new_tour() is called multiple times; once for each
                //      possible child-node/tour of the current node/tour

                // This is the only call-site to new_tour()

                // NOTE:  new_tour() does not modify curr_ind and curr
```

```

                // Can't change this w/o new_tour() changing
                TourElement cache_Tours_n_ind;

                if ((n_ind = new_tour( curr_ind, i, curr, &cache_Tours_n_ind ) )
                            == END_TOUR)
                    continue;
                // Changed:  I write it back out to GV_Tours[] here.
                //      This way, I get to keep a copy of the contents for
                //      access below (i.e., field lower_bound).
                GV_Tours[ n_ind ] = cache_Tours_n_ind;

                // No need to re-load curr_ind / curr since new_tour()
                //      only reads it, and does not modify it.

                // Actually, GS_MinTourLen is touched by set_best(), which
                //      is called by calc_bound()

                /*
                 * If it's shorter than the current best tour, or equal
                 * to the current best tour and we're reporting all min
                 * length tours, put it on the priority q.
                 */
                // n_ind is the index of the new tour being added
                //      Since I passed cache_Tours_n_ind into new_tour() above,
                //      I don't have to re-load it.  I've written it out above.
                newTour = &cache_Tours_n_ind;

                // Is there room on the PQ for this new tour?
                if (cache_PrioQLast >= MAX_NUM_TOURS-1)
                {
                    printf( "%d : Error : pqLast %d\n", MYPROC,cache_PrioQLast);
                    fflush(stdout);
                    exit(-1);
                }

                // Using a scoped behaviour above

                int * temp = (int *)GS_PrioQLast;
                pq = ++(*temp);
                cache_PrioQLast = pq;

                // Will only read-after-write, so no need to read first
                PrioQElement cache_PrioQ_cptr;  // No need to read
                int save_cptr_index = pq;
                cptr = &cache_PrioQ_cptr;

                cptr->index = n_ind;
                cptr->priority = newTour->lower_bound;

                // Write back to global store
                GV_PrioQ[ save_cptr_index ] = cache_PrioQ_cptr;

                /* Bubble the entry up to the appropriate level to maintain
                   the invariant that a parent is less than either of it's
                   children. */

                parent = pq >> 1;
                PrioQElement cache_PrioQ_pptr = GV_PrioQ[ parent ];
                int save_pptr_index = parent;
                pptr = &cache_PrioQ_pptr;

                while( ( pq > 1 ) &&
                        my_less_than( cptr, pptr, (TourElement*)GV_Tours) )
                {
                    /* cptr */ GV_PrioQ[ save_cptr_index ] =
                                /* pptr */ cache_PrioQ_pptr;
```

```
                    /* pptr */ GV_PrioQ[ save_pptr_index ] =
                                /* cptr */ cache_PrioQ_cptr; // Modified above

                    pq = parent;
                    // No need for:  cache_PrioQ_cptr = cache_PrioQ_pptr;
                    //  Just swap indices
                    save_cptr_index = save_pptr_index;

                    parent = pq >> 1;
                    cache_PrioQ_pptr = GV_PrioQ[ parent ];
                    save_pptr_index = parent;

                } // for-turned-while
            }
        } // for (all child nodes of current node)
    } // if

    } // End scope
} // split_tour


/*
 * find_solvable_tour():
 *
 * Used by both the normal TSP program (called by get_tour()) and
 * the RPC server (called by RPCserver()) to return the next solvable
 * (sufficiently short) tour.
 *
 */
// Called only by get_tour() (later in this file)

int find_solvable_tour( TourElement * currTour )
{
    register int curr, i, left, right, child, index;
    int priority, last;
    register PrioQElement *pptr, *cptr, *lptr, *rptr;

        /* Paul: Local caches of global variables  */
        PrioQElement cache_PrioQ_pptr; pptr = &cache_PrioQ_pptr; int pIndex;
        PrioQElement cache_PrioQ_cptr; cptr = &cache_PrioQ_cptr; int cIndex;
        PrioQElement cache_PrioQ_lptr; lptr = &cache_PrioQ_lptr; int lIndex;
        PrioQElement cache_PrioQ_rptr; rptr = &cache_PrioQ_rptr; int rIndex;

        int cache_PrioQLast;

    if ((int)GS_Done)
    {
        return(-1);
    }

    { // New scope
        NewBehaviour( GV_Tours, GVOwnerComputes, TourElement );
        GV_Tours.doParallel( GlobalMyTeam );

        NewBehaviour( GV_TourStack, GVOwnerComputes, int );
        GV_TourStack.doParallel( GlobalMyTeam );

        NewBehaviourScalar( GS_TourStackTop, GSOwnerComputes, int );
        NewBehaviourScalar( GS_PrioQLast, GSOwnerComputes, int );

        NewBehaviour( GV_PrioQ, GVOwnerComputes, PrioQElement );
        GV_PrioQ.doParallel( GlobalMyTeam );

    cache_PrioQLast = (int)GS_PrioQLast;
        // While the priority queue is not empty, find ready tours and
        //      put it on the TourStack
    for (; cache_PrioQLast != 0; )
```

```
    {
        cache_PrioQ_pptr = GV_PrioQ[ 1 ]; pIndex = 1;

        // The TourElement at 'curr' will be modified by the functions
        //      that it is passed to (e.g., split_tour())
        curr = pptr->index;

        // This is the tour from the top of the priority queue
        //      CacheTour.theTour is a *global* variable
        //      Should re-do this strategy in the future
        *currTour = GV_Tours[ curr ];

        if (pptr->priority >= (int)GS_MinTourLen)
        {
            /* We're done -- there's no way a better tour could be found. */
            GS_Done = 1;

            return(-1);
        } // if

        /* Bubble everything maintain the priority queue's heap invariant. */
        /* Move last element to root position. */
        cache_PrioQ_cptr = GV_PrioQ[ cache_PrioQLast ]; cIndex =cache_PrioQLast;

        // This actually makes the root of the tree be the last element of PQ
        cache_PrioQ_pptr = cache_PrioQ_cptr;    // Update cached copies
        GV_PrioQ[ pIndex ] = cache_PrioQ_cptr;  // Write back

        // Getting rid of last element, since I just moved it to root
        cache_PrioQLast = (int)GS_PrioQLast;    // Re-read.
        cache_PrioQLast--;
        GS_PrioQLast = cache_PrioQLast; // Write back

        /* Push previous last element down tree to restore heap structure. */
        for (i = 1; i <= (cache_PrioQLast >> 1); )
        {
            /* Find child with lowest priority. */
            left  = LEFT_CHILD(i);
            right = RIGHT_CHILD(i);

                cache_PrioQ_lptr = GV_PrioQ[ left ];  lIndex = left;
                cache_PrioQ_rptr = GV_PrioQ[ right ]; rIndex = right;

                // Pick the child branch to bubble down, if need (likely)
            if (left == cache_PrioQLast ||
                my_less_than(lptr,rptr,(TourElement*)GV_Tours))
            {
                child = left;
                cache_PrioQ_cptr = cache_PrioQ_lptr;
                cIndex = lIndex;
            }
            else {
                child = right;
                cache_PrioQ_cptr = cache_PrioQ_rptr;
                cIndex = rIndex;
            }

            /* Exchange parent and child, if necessary. */
            if (my_less_than(cptr,pptr,(TourElement*)GV_Tours))
            {
                /* glob->PrioQ[child] has lower prio than its parent */
                /*       switch 'em. */

                /* Swap parent and child in globally-shared memory */

                // Write back child as new parent
                GV_PrioQ[ pIndex ] = cache_PrioQ_cptr;  // Write back
```

```
                // Write back parent as new child
                GV_PrioQ[ cIndex ] = cache_PrioQ_pptr;  // Write back

                i = child;
                pIndex = i;

            } // if (need to exchange parent and child)
            else break;
        } // for (restore heap invariant)

        last = currTour->last;          // How long is this tour?

        /* If we're within 'NodesFromEnd' nodes of a complete tour, find
           minimum solutions recursively.  Else, split the tour. */
        if (last < TspSize || last < 1) {
            if (last >= (TspSize - NodesFromEnd - 1))
            {
                GV_Tours[ curr ] = *currTour;
                return(curr);
            } // if (within solve direct threshold)
                // Else:  Tour is too big to solve directly, so split it
            else
            {

        /* Paul:  The only place split_tour() is called. */
        /* Paul:  NOTE:  split_tour() does NOT modify curr/CacheTour.theTour */

                split_tour(curr, currTour );
                                /* The current tour is too long, */

                // NOTE:  We exit the if-else with more code to execute below.
                //        That code will free-up the tour we just split.
                //        Recall that splitting a tour puts all possible longer
                //        versions of the tour into the list, thus we no
                //        longer need this shorter tour.

            } // else (split the tour)

        /* Paul:  PrioQLast was changed in split_tour() since it */
        /* Paul:  added the new tours (multiple) to the priority queue */
            cache_PrioQLast = (int)GS_PrioQLast;
        }                               /* to solve now, break it up.    */
        else {
            /* Handle bogus tour index. */
            MakeTourString(TspSize, ( (TourElement)GV_Tours[curr] ).prefix);

            PrintTourString( "bogus tour index/find_solvable_tour" );
        } // else

        // If I have reached this point, then the current tour (cour)
        //        *must* have been split.  This means that this tour is no
        //        longer needed since it has been subsumed by its offspring,
        //        which are longer tours.
    {
        int temp = (int)GS_TourStackTop + 1;
        GS_TourStackTop = temp;         // Write back
        GV_TourStack[ temp ] = curr;
    }

    } // for (PQ not empty)

    } // End scope

    /* Ran out of candidates - DONE! */
    GS_Done = 1;
    return(-1);
} // find_solvable_tour
```

```
// Only called by Worker().  The input parameter 'curr' is non-"-1" iff
//      it was already previously solved by a call to recursive_solve().
//      See the loop in Worker() for details.

int get_tour(register int curr, TourElement * currTour )
{
        // If curr != -1, that means that recursive_solve() has already
        //      finished with that tour, so we can safely free it.
        //      See the loop in Worker() for details.
    if (curr != -1)
    {

    { // New scope
        NewBehaviourScalar( GS_TourStackTop, GSOwnerComputes, int );

        NewBehaviour( GV_TourStack, GVOwnerComputes, int );
        GV_TourStack.doParallel( GlobalMyTeam );

        int temp = (int)GS_TourStackTop + 1;
        GS_TourStackTop = temp;         // Write back
        GV_TourStack[ temp ] = curr;
    } // End scope

    } // if

    curr = find_solvable_tour( currTour );

    return(curr);
} // get_tour
```

B.4.5 read_tsp.C

```
#include <stdio.h>
#include        "tsp.h"
#include        "tsp.H"

int ExpectedAnswer;

#define COMM_LEN        512
char comments[ COMM_LEN ];

void read_comments( FILE * fp )
{
        int inC;
        char * start = comments;
        char * stop = &( comments[ COMM_LEN - 2 ] );
        do
        {
                inC = fgetc( fp );
                *start++ = (char)inC;
        }
        while( ( inC != EOF ) && ( start < stop ) );
        *( start - 1 ) = '\0';
        comments[ COMM_LEN - 1 ] = '\0';        // Make sure null terminated
} // read_comments

void print_comments()
{
        printf( "%s\n", comments );
```

```
} // print_comments

int theExpectedAnswer()
{
        return( ExpectedAnswer );
} // theExpectedAnswer

char *read_int(char * char_ptr, int * wt)
{
    char *temp_ptr;

    temp_ptr = char_ptr;

    while (*temp_ptr == ’ ’ || *temp_ptr == ’\t’)
        temp_ptr++;

    if (*temp_ptr == ’\n’ || *temp_ptr == ’\0’)
        return(NULL);

    while (*temp_ptr != ’ ’ && *temp_ptr != ’\t’ && *temp_ptr != ’\n’)
        temp_ptr++;

    *temp_ptr = ’\0’;

    *wt = atoi(char_ptr);

    return(temp_ptr + 1);
} // read_int


int read_tsp( char * file)
{
    FILE *fp;
    char *scan_ptr, line[81], fname[80];
    int wt, i ,j, fully_connected = 1;

    strcpy(fname,"tspfiles/tspfile");
    strcat(fname,file);
    if ((fp = fopen(fname, "r")) == NULL) {
        printf("Cannot open TSP file: %s\n", fname);
        exit(0);
    }

    fgets(line, 80, fp);

    TspSize = atoi(line);
    if (TspSize > MAX_TOUR_SIZE) {
        fprintf(stderr,"Error: Problem size (%d) larger than maximum (%d).\n",
                TspSize, MAX_TOUR_SIZE);
        fprintf(stderr,"ABORTING.\n");
        exit(-1);
    }

    for (i = 0; i < TspSize; i++) {
        scan_ptr = fgets(line, 80, fp);
        j = 0;
        while (scan_ptr = read_int(scan_ptr, &wt)) weights[i][j++] = wt;
    }

        fscanf( fp, "%d", &ExpectedAnswer );  /* Read in the expected answer */
        printf( "The expected answer is %d / %d\n",
                theExpectedAnswer(), ExpectedAnswer );

        read_comments( fp );
        fclose( fp );

        print_comments();
```

```

    return(TspSize);
} // read_tsp
```

B.4.6 solve.C

```
#include <stdio.h>

#include "tsp.h"
#include "tsp.H"

unsigned        CurDist, PathLen;
int             Visit[MAX_TOUR_SIZE], Path[MAX_TOUR_SIZE];
int             visitNodes;

extern void visit_nodes(int from);

/*
 *   recursive_solve(curr_ind)
 *
 *      We're now supposed to recursively try all possible solutions
 *      starting with the current tour.  We do this by copying the
 *      state to local variables (to avoid unneeded conflicts) and
 *      calling visit_nodes to do the actual recursive solution.
 *
 */

void recursive_solve(int index, TourElement * currTour )
{
    register unsigned i, j;
    register TourElement *curr;
    curr = currTour;                    // Passed in as a parameter

    CurDist = curr->prefix_weight;
    PathLen = curr->last + 1;

        /* Paul:  Visit[] is *global* variable above */
    for (i = 0; i < TspSize; i++) Visit[i] = 0;

    for (i = 0; i < PathLen; i++) {
        Path[i] = curr->prefix[i];
        Visit[Path[i]] = 1;
    }

    if (PathLen > TspSize) {
        printf( "%d : Error : Pathlen: %d\n", MYPROC, PathLen);
        fflush(stdout);
        exit(0);
    }

    visit_nodes(Path[PathLen-1]);
} // recursive_solve


/*
 *   visit_nodes()
 *
 *       Exhaustively visits each node to find Hamilton cycle.
 *       Assumes that search started at node from.
 *
 */
void visit_nodes(int from)
{
    register int i, j;
    register int dist, last;
```

```


    visitNodes++;

    for (i = 1; i < TspSize; i++) {
        if (Visit[i]) continue; /* Already visited. */
        if ((dist = weights[from][i]) == 0) continue; /* Not connected. */
        if (CurDist + dist >  (int)GS_MinTourLen )
                continue; /* Path too long. */

        /* Try next node. */
        Visit[i] = 1;
        Path[PathLen++] = i;
        CurDist += dist;

        if (PathLen == TspSize) {
            /* Visiting last node - determine if path is min length. */
            if ((last = weights[i][StartNode]) != 0 &&
                (CurDist += last) < (int)GS_MinTourLen )
            {
        /* Paul:  Function changes values */
                set_best(CurDist, Path);
            }
            CurDist -= last;
        } /* if visiting last node */
        else if (CurDist < (int)GS_MinTourLen ) visit_nodes(i);/* Visit on. */

        /* Remove current try and loop again at this level. */
        CurDist -= dist;
        PathLen--;
        Visit[i] = 0;
    } // for
} // visit_nodes


/*
 *  Add up min edges connecting all unconnected vertices (AHU p. 331-335)
 *  At some point, we may want to use MST to get a lower bound. This
 *  bound should be higher (and therefore more accurate) but will take
 *  longer to compute.
 */

// Called by tsp_orig_main_2()/MASTER in main.C
// Called by new_tour()/MASTER in tours.C

int calc_bound( int curr_index, TourElement * currInput )
{
    register int i, j, wt, wt1, wt2;
    register TourElement *curr;
    curr = currInput;           // A parameter of the function

    /*
     * wt1: the value of the edge with the lowest weight from the node
     *      we're testing to another unconnected node.
     * wt2: the value of the edge with the second lowest weight
     */

    /* if we have one unconnected node */
    if (curr->last == (TspSize - 2)) {

    { // New scope
        NewBehaviour( GV_TourStack, GVOwnerComputes, int );
        GV_TourStack.doParallel( GlobalMyTeam );

        NewBehaviourScalar( GS_TourStackTop, GSOwnerComputes, int );

        for (i = 0; i < TspSize; i++) {
```

```
            if (!(curr->conn & (1<<i))) {
                /* we have found the one unconnected node */
                curr->prefix[TspSize-1] = i;
                curr->prefix[TspSize] = StartNode;

                /* add edges to and from the last node */
                curr->prefix_weight+= weights[curr->prefix[TspSize-2]][i] +
                                      weights[i][curr->prefix[StartNode]];

                if (curr->prefix_weight < (int)GS_MinTourLen)
                {
        /* Paul:  set_best() (in tours.C) modifies MinTourLen and MinTour[] */

                    /* Store our new best path and its weight. */
                    set_best(curr->prefix_weight, curr->prefix);
                }

        // GV_TourStack keeps track of the empty entries/tours in GV_Tours[]
        //      I am freeing this partially evaluated tour because it's current
        //      length is already larger than the best complete tour so far

                /* De-allocate this tour so someone else can use it */
                curr->lower_bound = BIGINT;

                int temp = (int)GS_TourStackTop + 1;
                GS_TourStackTop = temp;         // Write back
                GV_TourStack[ temp ] = curr_index;

                // Assumes caller will writeback the changes
                return(END_TOUR);
            } // if
        } // for

    } // End scope

    } // if

    curr->mst_weight = 0;

    /*
     * Add up the lowest weights for edges connected to vertices
     * not yet used or at the ends of the current tour, and divide by two.
     * This could be tweaked quite a bit.  For example:
     *   (1) Check to make sure that using an edge would not make it
     *       impossible for a vertex to have degree two.
     *   (2) Check to make sure that the edge doesn't give some
     *       vertex degree 3.
     */

    if (curr->last != TspSize - 1) {
        for (i = 0; (i < TspSize); i++) {
            if (curr->conn & 1<<i) continue;
            for (j = 0, wt1 = wt2 = BIGINT; j < TspSize; j++) {
                /* Ignore j's that are not connected to i (weights[i][j]==0), */
                /* or that are already in the tour and aren't either the      */
                /* first or last node in the current tour.                    */
                wt = weights[i][j];
                if (!wt || (curr->conn&(1<<j)&&(j != curr->prefix[0]) &&
                                               (j != curr->prefix[curr->last])))
                    continue;

                /* Might want to check that edges go to unused vertices */
                if (wt < wt1) {
                    wt2 = wt1;
                    wt1 = wt;
                }
                else if (wt < wt2) wt2 = wt;
```

```
            }

            /* At least three unconnected nodes? */
            if (wt2 != BIGINT) curr->mst_weight += (wt1 + wt2) >> 1;
            /* Exactly two unconnected nodes? */
            else if (wt1 != BIGINT) curr->mst_weight += wt1;
        }
        curr->mst_weight += 1;
    }

    curr->lower_bound = curr->mst_weight + curr->prefix_weight;

    // Assumes caller will write back the changes to curr_index / curr
    return(curr_index);
} // calc_bound
```

B.4.7 tours.C

```
#include <stdio.h>

#include "tsp.h"
#include "tsp.H"

char tour_str[256];

/*
 * new_tour():
 *
 *    Create a new tour structure given an existing tour structure and
 *  the next edge to add to the tour.  Returns the index of the new structure.
 *
 */
// My caller, split_tour() is responsible for inserting it properly into the PQ

// I am only called within split_tour() (pq.C)
int new_tour(int prev_index, int move, TourElement * prevTour,
                TourElement * newTour )
{
    int index, i;
    register TourElement *curr, *prev;
    int rval;

    { // New scope
        NewBehaviour( GV_TourStack, GVOwnerComputes, int );
        GV_TourStack.doParallel( GlobalMyTeam );

        NewBehaviourScalar( GS_TourStackTop, GSOwnerComputes, int );

        // NOTE:  TourStack keeps track of blank tours/entries in Tour[]

    /* get index of a blank tour */

    int cache_GS_TourStackTop = (int)GS_TourStackTop;
    if ( cache_GS_TourStackTop >= 0 )
    {
        index = GV_TourStack[ cache_GS_TourStackTop ];

        // Write back.  Do the post --
        GS_TourStackTop = cache_GS_TourStackTop - 1;
    } // if
    else
    {
        printf( "%d : Error: TourStackTop: %d\n", MYPROC, (int)GS_TourStackTop);
        fflush(stdout);
        exit(-1);
```

```
    }

        // NOTE:  'index' is where the newly created tour will go
        // Overwrites-only.  Read-after-write only (don't read-in).
        curr = newTour;                 // This is now a function parameter
        prev = prevTour;                // This is now a function parameter

        /* Paul:  About to modify glob->Tours[index] */

    for (i = 0; i < TspSize; i++) {
        curr->prefix[i] = prev->prefix[i];
    }
    curr->conn = prev->conn;    // Moved here from inside the loop

    curr->last = prev->last;
    curr->prefix_weight = prev->prefix_weight +
                          weights[curr->prefix[curr->last]][move];
    curr->prefix[++(curr->last)] = move;

    /* add our new move */
    curr->conn |= 1 << move;

    /* calculate and return the new lower bound */
        rval = calc_bound( index, curr );

        // GV_Tours[ index ] is now, it is written out by split_tour()

    } // End scope

        return( rval );
} // new_tour


/*
 * set_best():
 *
 *  Set the global 'best' value.
 *
 */
void set_best( int best, int * path)
{
    int i;

    if (best >= (int)GS_MinTourLen)
    {
        return;         /* Changed to return void.  Should be ok, right? */
    }

    if (best < (int)GS_MinTourLen)
    {
        GS_MinTourLen = best;

    { // New scope
        NewBehaviour( GV_MinTour, GVReleaseC, int );

        for (i = 0; i < TspSize; i++)
        {
                GV_MinTour[i] = path[i];
        }
    } // End scope

    } // if
} // set_best


/*
 *  MakeTourString(len, path):
```

```
 *
 *  Make a string for printing that describes the tour passed via len and path.
 *
 */
void MakeTourString(int len, int * path)
{
    int i, j;

    for (i = j = 0; i < len; i++) {
        sprintf((tour_str + j), "%1d - ", (int)path[i]);
        if ((int)path[i] >= 10) j += 5; /* Two digit number. */
        else j += 4;
    }
    sprintf((tour_str + j), "%1d\0", (int) path[i]);

} // MakeTourString


void PrintTourString( char * annotation )
{
        printf( "%d : %s : %s\n ", MYPROC, annotation, tour_str );
} // PrintTourString
```

Appendix C

Example TreadMarks Program

The following segments of code have been partially cleaned up for presentation. They are included to give the reader a flavour of the programming system; a lot of the debugging and timing code have been removed. However, it should be noted that the code and the programming style remain raw since, as the saying goes: you never finish a program, you only stop working on it.

C.1 Matrix Multiplication

C.1.1 main.C

```
/*
======================================================================

        Matrix Multiplication Using TreadMarks

======================================================================
*/

#include "TmkCompat.H"          /* Mini library to make porting easier */

#include <stdio.h>
#include <memory.h>             /* For memcpy() */

#include "multidim.H"           /* For 2-D index object:  Index2D<> */

#define NUM_RUNS        6
Index2D<int>    idx;

/*

        Globals

*/
int     * XnumPEPtr;            /* Globals to pass info to all PEs */
#define XnumPE          (*XnumPEPtr)
int     * XmatrixSizePtr;
#define XmatrixSize     (*XmatrixSizePtr)

int     * MatrixA;              /* Don't need access macros for arrays */
int     * MatrixB;
```

```
int     * MatrixC;

/* Note:  First run not used in averaging */

int Seed[ 11 ] = { 2,   2, 3, 4, 5, 6,   2, 3, 4, 5, 6 };

int     * RunNumPtr;            /* Current run number */
#define RunNum          (*RunNumPtr)
int PrivateRunNum;              /* Easier to emulate Aurora code with this */
int     * DebugPtr;             /* Flag:  Debug */
#define Debug           (*DebugPtr)

TimeObj ParTimer    ( "Total time in PARALLEL:" );

double RunTimes[ NUM_RUNS ];    /* Save run times for averaging */

extern void setUp( int argc, char ** argv, char ** envp );
extern void * matrixmultiply( void * arg );
extern void mmultiply( int * a, int * b, int * c, int m, int n, int p,
                int startRow, int endRow );

extern void createData( const int myId, const int theStart, const int theEnd,
                int * theMatrix, const int randSeed, char * msg = "" );

// Common function to parallel and sequential
inline int dotProduct( int * a, int * b, const int j, const int n )
{
        register int sum = 0, k;

        for( k = 0; k < n; k++ )
        {
                sum += a[ k ] * b[ idx.i( k, j ) ];
        }

        return( sum );
}



// PARALLEL and SEQUENTIAL
/* mmultiply:  Multiplies matrices in parameters.  Can be used
   for a variety of different global variables, depending on parameters. */

void mmultiply( int * a, int * b, int * c, int m, int n, int p,
        int startRow, int endRow )
{
        int     i, j, k;                /* Loop indices */

        for( i = startRow; i < endRow; i++ )
        {
                for( j = 0; j < p; j++ )
                {
                        c[ idx.i( i,  j ) ] =
                                dotProduct( &a[ idx.i( i, 0 ) ], b,  j, n );
                }
        }
} /* mmultiply */



/*
------------------------------

NAME:   Main

------------------------------
*/
```

```

int main( int argc, char ** argv, char ** envp )
{
        int     i;

        Tmk_startup( argc, argv );

        setUp( argc, argv, envp );

        N_BARRIER( "end setup : start matrixmultiply" );

        /* Do the multiply (within a run-loop) */
        matrixmultiply( NULL );

        N_BARRIER( "About to finalize and exit..." );

        Tmk_exit( 0 );

        return 0;
} // main



void setUp( int argc, char ** argv, char ** envp )
{
MASTER
{
        // We use pointers to ints (instead of just ints for constants)
        //      to increase flexibility to change the values on-the-fly
        //      if needed, down the road.

        My_Tmk_malloc( XnumPEPtr, int, 1 );             // See TmkCompat.H
        My_Tmk_malloc( XmatrixSizePtr, int, 1 );        // See TmkCompat.H
        My_Tmk_malloc( RunNumPtr, int, 1 );             // See TmkCompat.H
        My_Tmk_malloc( DebugPtr, int, 1 );              // See TmkCompat.H

        int     i, j, k;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */

        /* Process command line */

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n----------------------------------------\n" );
        printf( "Matrix Multiplication Using TreadMarks\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- How large of a matrix? */

        XmatrixSize = atoi( argv[ 2 ] );  /* Get from input line: arg 2 */
        printf( "\tSquare matrix size is:  %d x %d\n",
                (int)XmatrixSize, (int)XmatrixSize );

        /* Set up indexing object */

        idx.set( (int)XmatrixSize, (int)XmatrixSize );

        /* Everybody must call method owners().  It sets-up internal values. */
        idx.owners( (int)XnumPE );              // Divy up the rows

        /* Now that we've initialized it, tell everybody */
        Tmk_distribute( (char *)& idx, sizeof( idx ) );

```

```
        My_Tmk_malloc( MatrixA, int, idx.size() );      // See TmkCompat.H
        My_Tmk_malloc( MatrixB, int, idx.size() );      // See TmkCompat.H
        My_Tmk_malloc( MatrixC, int, idx.size() );      // See TmkCompat.H

} // MASTER
END_MASTER;
N_BARRIER( "setUp" );

} // setUp




void * matrixmultiply( void * arg )
{
        int     i, j, k;
        int     m, n, p;                        // Matrix dimensions
        int     start, end;


        m = n = p = (int)XmatrixSize;
        idx.set( (int)XmatrixSize, (int)XmatrixSize );

BARRIER;                /* Wait for all PEs to start-up */

N_BARRIER( "Start All Runs" );

        /* Do NUM_RUNS number of test, and average statistics */

        for( RunNum = 0; RunNum < NUM_RUNS; )
        {
                PrivateRunNum = RunNum; // Get local copy

                /* Master will increment RunNum later */

                /* Create data in parallel */

N_BARRIER( "Start Create Data: A" );

                i = _IPROC;
                int startRow    = idx.myBeginRow( i );
                int endRow      = idx.myEndRow( i );

                // Convert row number to row-major 1-D index
                // Assume [ start, end ) is the desired semantics
                start   = idx.i( startRow, 0 );
                end     = idx.i( endRow, 0 );

                createData( _IPROC, start, end, MatrixA, Seed[ PrivateRunNum ],
                        "MatrixA Partition" );

N_BARRIER( "Start Create Data: B" );

                // Note:  multiplicative factor of 7
                createData( _IPROC, start, end,
                                        MatrixB, 7 * Seed[ PrivateRunNum ],
                        "MatrixB Partition" );

N_BARRIER( "End Create Data" ); // Wait for data to be created

                /*
                        ----- PARALLEL -----
                */
                if ( (int)XnumPE > 1 )
                {
N_BARRIER( "Start Parallel : C = A x B" );

```

```
                    { TimeScope timer( ParTimer, false );

                        mmultiply( MatrixA, MatrixB, MatrixC, m, n, p,
                                idx.myBeginRow( _IPROC ),
                                idx.myEndRow( _IPROC ) );

N_BARRIER( "End C = A x B : Start B = A x C" );

                        mmultiply( MatrixA, MatrixC, MatrixB, m, n, p,
                                idx.myBeginRow( _IPROC ),
                                idx.myEndRow( _IPROC ) );
                    } // End timer

N_BARRIER( "End B = A x C" );
                    } // End ParTimer timer
                }
BARRIER;

MASTER
{
                /* Done multiplying.  Calc statistics. */
                printf( "Done!  Run Number %d.\n", (int)RunNum );

                ParTimer.print( "\n" );
                if( (int)RunNum > 0 )
                        ParTimer.addSample();

} // MASTER
END_MASTER;

BARRIER;

MASTER
{
                RunNum++;                       /* Ready to do next run */
} // MASTER
END_MASTER;

                fflush( stdout );
N_BARRIER( "End RunNum" );
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );

MASTER
{
        printf( "\n" );

        printf( "%d PEs / %d x %d / Tmk Version of MM2\n\n",
                (int)XnumPE, (int)XmatrixSize, (int)XmatrixSize );

        ParTimer.stats( "Avg time (FINAL)\n" );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        return NULL;
} /* matrixmultiply */
```

```
// Create data for my co-located data via owners-computes
//      The values are random (via srand()) and the seed if a function
//      of the parameter 'randSeed' and my rank in my team.
void createData( const int myId,
                 const int theStart,
                 const int theEnd,
                 int * theMatrix,
                 const int randSeed,
                 char * msg
)
{

        // Extra indentation to better match formatting in Aurora version
                int i = myId;
                int start = theStart;
                int end   = theEnd;

                // One-beyond (instead of "end - 1") semantics now

                /* Fill [ start , end ) with data */

                srand( ( i + 1 ) * randSeed );

                int theVal, k;
                for( k = start; k < end; k++ )
                {
                        theVal = rand();        /* Create data */
                        theVal %= 31;   // Limit the range to limit overflow

                        theMatrix[ k ] = theVal;
                }
} // createData
```

C.1.2 TmkCompat.H

```
//================================================================
//
//      TmkCompat.H
//
//      Macros, etc. to make it easier to port between Aurora and TreadMarks
//
//================================================================

#ifndef _TMKCOMPAT_H_
#define _TMKCOMPAT_H_

#include <Tmk.h>
#define bool    int
#define true    1
#define false   0

#define THEMASTER       ( Tmk_proc_id == 0 )

#define MASTER          if( THEMASTER )
#define END_MASTER      // Nothing
#define _IPROC          ( Tmk_proc_id )
#define _NPROCS         ( Tmk_nprocs )
#define MAX_LOCALS      ( TMK_NPROCS )

// The macros were relative to nodes in Aurora (i.e., P_RTS)
//      But that is not the case under TreadMarks
#define RTS_NPROCS      ( _NPROCS )
#define MYPROC          ( _IPROC )
#define FIRST           ( MYPROC == 0 )
#define LAST            ( MYPROC == ( RTS_NPROCS - 1 ) )
```

```
#define FIRST_LAST      ( FIRST || LAST )


        // Use the mod function to better detect barrier mismatch problems
#ifndef BENCH_AURORA
#define BARRIER                                                     \
    {                                                               \
        MASTER                                                      \
        {                                                           \
                printf( "C%d : %s\n", __LINE__, __FILE__ );         \
        }                                                           \
        Tmk_barrier( ( __LINE__ % ( TMK_NBARRIERS - 1 ) ) );        \
    }

#define N_BARRIER( NAME )                                           \
    {                                                               \
        MASTER                                                      \
        {                                                           \
                printf( "C%d : %s  %s\n", __LINE__, __FILE__, NAME ); \
        }                                                           \
        Tmk_barrier( ( __LINE__ % ( TMK_NBARRIERS - 1 ) ) );        \
    }

#else   /* else NOT BENCH_AURORA */

#define BARRIER                                                     \
    {                                                               \
        Tmk_barrier( ( __LINE__ % ( TMK_NBARRIERS - 1 ) ) );        \
    }

#define N_BARRIER( NAME )                                           \
    {                                                               \
        Tmk_barrier( ( __LINE__ % ( TMK_NBARRIERS - 1 ) ) );        \
    }
#endif /* else NOT BENCH_AURORA */


#include <stdio.h>
#include "stats.H"

// NAME is the global variable name     (must be a pointer)
// TYPE is the global variable type     (must *not* be a pointer type)
// SIZE is the global variable set size (can be array)

#define My_Tmk_malloc( NAME, TYPE, SIZE )                           \
    {                                                               \
        printf( "Shared memory allocate+distribute : '" # NAME      \
                "' type = " # TYPE ", elements = %d\n", SIZE );     \
        NAME = (TYPE *)Tmk_malloc( SIZE * sizeof( * NAME ) );       \
        Tmk_distribute( (char *)& NAME, sizeof( NAME ) );           \
    }
#endif // _TMKCOMPAT_H_
```

Appendix D

Example Message-Passing Program

The following segments of code have been partially cleaned up for presentation. They are included to give the reader a flavour of the programming system; a lot of the debugging and timing code have been removed. However, it should be noted that the code and the programming style remain raw since, as the saying goes: you never finish a program, you only stop working on it.

D.1 Matrix Multiplication (MPICH)

D.1.1 main.C

```
/*
=======================================================================

        Matrix Multiplication Using MPI

=======================================================================
*/

#include "aurora2mpi.H"         /* For compatibility functions */

#include <stdio.h>
#include <memory.h>             /* For memcpy() */

#include "multidim.H"           /* For 2-D index object:  Index2D<> */

#define NUM_RUNS        6

Index2D<int>    idx;

/*

        Globals

*/
int     XnumPE;                 /* Globals to pass info to all PEs */
int     XmatrixSize;

int     * MatrixA;
int     * MatrixB;
int     * MatrixC;
```

```
int     * MatrixTemp;


/* Note:  First run not used in averaging */

int Seed[ 11 ] = { 2,   2, 3, 4, 5, 6,   2, 3, 4, 5, 6 };

int RunNum;                     /* Current run number */
int PrivateRunNum;              /* Easier to emulate Aurora code with this */
int Debug;                      /* Flag:  Debug */

TimeObj ParTimer    ( "Total time in PARALLEL:" );

double RunTimes[ NUM_RUNS ];    /* Save run times for averaging */

extern void setUp( int argc, char ** argv, char ** envp );
extern void * matrixmultiply( void * arg );
extern void mmultiply( int * a, int * b, int * c, int m, int n, int p,
        int startRow, int endRow );


extern void createData( const int myId, const int theStart,
        const int theEnd, int * theMatrix, const int randSeed,
        char * msg = "" );


// Common function to parallel and sequential
inline int dotProduct( int * a, int * b, const int j, const int n )
{
        register int sum = 0, k;

        for( k = 0; k < n; k++ )
        {
                sum += a[ k ] * b[ idx.i( k, j ) ];
        }

        return( sum );
}




// PARALLEL and SEQUENTIAL
// Taken from the Tmk version
/* mmultiply:  Multiplies matrices in parameters.  Can be used
   for a variety of different global variables, depending on parameters. */

void mmultiply( int * a, int * b, int * c, int m, int n, int p,
        int startRow, int endRow )
{
        int     i, j, k;                /* Loop indices */

        for( i = startRow; i < endRow; i++ )
        {
                for( j = 0; j < p; j++ )
                {
                        c[ idx.i( i,  j ) ] =
                                dotProduct( &a[ idx.i( i, 0 ) ], b,  j, n );
                }
        }
} /* mmultiply */

/*
------------------------------

NAME:   Main

```

```
------------------------------
*/

int main( int argc, char ** argv, char ** envp )
{
        int     i;

        MyMPIInit( &argc, &argv );

        // Under MPI, assume that everybody runs setUp() SPMD-style
        //      This is different than Aurora (but same as Tmk, see code there)

        setUp( argc, argv, envp );

        N_BARRIER( "end setup : start matrixmultiply" );

        // Enter SPMD-style
        matrixmultiply( NULL );

        N_BARRIER( "About to finalize and exit..." );

        MyMPIFinalize();

        return 0;
} // main



void setUp( int argc, char ** argv, char ** envp )
{
MASTER
{
        int     i, j, k;
        int     currentArg;     /* Current command line argument */
        char    * cp;           /* For reading line arguments */

        /* Process command line */

        /* Argument 1 -- How many processors? */

        XnumPE = atoi( argv[ 1 ] );     /* Get from input line: arg 1 */
        printf( "\n\n----------------------------------------\n" );
        printf( "Matrix Multiplication Using MPI\n\n" );
        printf( "Environment report:\n" );
        printf( "\tCompiler version:  %x\n", __xlC__ );
        printf( "\tNumber of processors is:  %d\n", (int)XnumPE );

        /* Argument 2 -- How large of a matrix? */

        XmatrixSize = atoi( argv[ 2 ] );        /* Get from input line: arg 2 */
        printf( "\tSquare matrix size is:  %d x %d\n",
                (int)XmatrixSize, (int)XmatrixSize );

        /* Set up indexing object */

        idx.set( (int)XmatrixSize, (int)XmatrixSize );

        /* Everybody must call method owners().  It sets-up internal values. */
        idx.owners( (int)XnumPE );

} // MASTER
END_MASTER;
N_BARRIER( "setUp : end master" );

        /* This phase pushes some global data from the master to non-masters */

        MyMPIDistribute( & idx, sizeof( idx ), "idx" );
```

```

        /* NOTE:  These next 3 values are done in Tmk using pointers to
                shared variables instead */
        MyMPIDistribute( & XnumPE, sizeof( XnumPE ), "XnumPE" );
        MyMPIDistribute( & XmatrixSize, sizeof( XmatrixSize ), "XmatrixSize" );
        MyMPIDistribute( & Debug, sizeof( Debug ), "Debug" );

N_BARRIER( "setUp : end MyMPIDistribute" );

        /* Allocate space for the data */

        /* Under MPI, just allocate an entire, dense matrix */

        MatrixA = (int *)valloc( idx.bytes() );
        MatrixB = (int *)valloc( idx.bytes() );
        MatrixC = (int *)valloc( idx.bytes() );

        /* Need a temporary buffer since, under MPI, we cannot have a src
                and dest buffer be the same.  However, we will swap pointers
                between this buffer and another instead of doing a "copy"
                of buffer contents. */
        MatrixTemp = (int *)valloc( idx.bytes() );

} // setUp




void * matrixmultiply( void * arg )
{
        int     i, j, k;
        int     m, n, p;                /* Matrix dimensions */
        int     start, end;             /* Rows of a 2-D matrix */


        m = n = p = (int)XmatrixSize;
        idx.set( (int)XmatrixSize, (int)XmatrixSize );

BARRIER;                /* Wait for all PEs to start-up */

N_BARRIER( "Start All Runs" );

        /* Do NUM_RUNS number of test, and average statistics */

        for( RunNum = 0; RunNum < NUM_RUNS; )
        {
                PrivateRunNum = RunNum; /* Get local copy */

                /* Master will increment RunNum later */

                /* Create data in parallel */

N_BARRIER( "Start Create Data: A" );

                i = _IPROC;
                int startRow    = idx.myBeginRow( i );
                int endRow      = idx.myEndRow( i );

                // Convert row number to row-major 1-D index
                // Assume [ start, end ) is the desired semantics
                start   = idx.i( startRow, 0 );
                end     = idx.i( endRow, 0 );

                createData( _IPROC, start, end, MatrixA, Seed[ PrivateRunNum ],
                        "MatrixA Partition" );

N_BARRIER( "Start Create Data: B" );
```

```

                // Note:  multiplicative factor of 7
                createData( _IPROC, start, end,
                                        MatrixB, 7 * Seed[ PrivateRunNum ],
                        "MatrixB Partition" );

N_BARRIER( "End Create Data" ); // Wait for data to be created

                /*
                        ----- PARALLEL -----
                */
                if ( (int)XnumPE > 1 )
                {
N_BARRIER( "Start Parallel : C = A x B" );

                    { TimeScope timer( ParTimer, false );

                        // Manual/explicit preparation for owners-computes

                        int startRow    = idx.myBeginRow( _IPROC );
                        int endRow      = idx.myEndRow( _IPROC );

                        // Convert row number to row-major 1-D index
                        // Assume [ start, end ) is the desired semantics
                        int start       = idx.i( startRow, 0 );
                        int end         = idx.i( endRow, 0 );

                        // Unique to MPI version, need to *explicitly* gather
                        //      contents of Matrix B before multiplying
                        // SPMD function.  Gather into MatrixB.  Note:  we
                        //      cannot use same buffer as src and dest.

                        MyMPIAllgather( MatrixB /* src */, MatrixTemp,
                                start, end, "MatrixB-Temp" );

                        // Swap the pointers to buffers MatrixB and MatrixTemp
                        int * tempPtr = MatrixTemp;
                        MatrixTemp = MatrixB;
                        MatrixB = tempPtr;

                        mmultiply( MatrixA, MatrixB, MatrixC, m, n, p,
                                idx.myBeginRow( _IPROC ),
                                idx.myEndRow( _IPROC ) );

N_BARRIER( "End C = A x B : Start B = A x C" );

                        // Explicitly gather contents of Matrix C

                        MyMPIAllgather( MatrixC /* src */, MatrixTemp,
                                start, end, "MatrixC-Temp" );

                        // Swap the pointers to buffers MatrixC and MatrixTemp
                        tempPtr = MatrixTemp;
                        MatrixTemp = MatrixC;
                        MatrixC = tempPtr;

                        mmultiply( MatrixA, MatrixC, MatrixB, m, n, p,
                                idx.myBeginRow( _IPROC ),
                                idx.myEndRow( _IPROC ) );

                        // Return pointers to buffers to initial state

                        // Swap the pointers to buffers MatrixB and MatrixC
                        tempPtr = MatrixB;
                        MatrixB = MatrixC;
                        MatrixC = tempPtr;
                        // Matrix B is now back in place
```

```

                        // Swap the pointers to buffers MatrixC and MatrixTemp
                        tempPtr = MatrixC;
                        MatrixC = MatrixTemp;
                        MatrixTemp = tempPtr;
                        // Matrix B  and MatrixTemp are now back in place
N_BARRIER( "End B = A x C" );
                    } // End timer
                }
BARRIER;

MASTER
{
                /* Done multiplying.  Calc statistics. */
                printf( "Done!  Run Number %d.\n", (int)RunNum );

                ParTimer.print( "\n" );
                if( (int)RunNum > 0 )
                        ParTimer.addSample();
} // MASTER
END_MASTER;

BARRIER;

MASTER
{
                RunNum++;                       /* Ready to do next run */
} // MASTER
END_MASTER;
BARRIER;

                fflush( stdout );

                /* Synchronized push of RunNum from the root outwards */
                MyMPIDistribute( & RunNum, sizeof( RunNum ), "RunNum" );

N_BARRIER( "End RunNum" );
        } /* for */

                fflush( stdout );
N_BARRIER( "End All Runs" );

MASTER
{
        printf( "\n" );
        printf( "%d PEs / %d x %d / MPI Version of MM2\n\n",
                (int)XnumPE, (int)XmatrixSize, (int)XmatrixSize );

        ParTimer.stats( "Avg time (FINAL)\n" );

        fflush( stdout );
} // MASTER
END_MASTER;
BARRIER;                        /* Early exiter kills other processes */

        return NULL;
} /* matrixmultiply */




// Create data for my co-located data via owners-computes
//      The values are random (via srand()) and the seed if a function
//      of the parameter 'randSeed' and my rank in my team.
void createData( const int myId,
                 const int theStart,
                 const int theEnd,
```

```
                 int * theMatrix,
                 const int randSeed,
                 char * msg
)
{
        // Extra indentation to better match formatting in Aurora version
                int i = myId;
                int start = theStart;
                int end   = theEnd;

                // One-beyond (instead of "end - 1") semantics now

                /* Fill [ start , end ) with data */

                srand( ( i + 1 ) * randSeed );

                int theVal, k;
                for( k = start; k < end; k++ )
                {
                        theVal = rand();        /* Create data */
                        theVal %= 31;   // Limit the range to limit overflow

                        theMatrix[ k ] = theVal;
                }
} // createData

#include "myconfig.H"                   // From libaurora/include
```

D.1.2 aurora2mpi.H

```
//================================================================
//
//      aurora2mpi.H
//
//      This file emulates and provides glue to make it easier
//      to port Aurora applications to MPI.
//
//================================================================

#ifndef _AURORA2MPI_H_
#define _AURORA2MPI_H_

// Max number of LVectors/AOs
const int MAX_LOCALS = 16;

// STL-compliant definitions
#define bool int
#define true 1
#define false 0

// Initialization
extern bool InitWasCalled;
extern int MyMPIInit( int * argcPtr, char *** argvPtr );
extern void MyMPIFinalize();

// Higher-level data exchange abstractions for SPMD programming
        // Use this like Tmk_distribute() in TreadMarks/Tmk
/*
        ** Not templated.  Sends bytes.
        MyMPIBroadcast  : 1-to-N        broadcast communication
        MyMPIDistribute : root/1-to-N   broadcast communication

        MyMPIGather     : N-to-1        gather communication,
                                        constant/fixed block-partitioning

```

```
        ** Templated.  Sends data elements.
        MyMPIAllgather  : N-to-N        all-to-all communication,
                                        arbitrary block-partitioning
*/
extern void MyMPIBroadcast( const int srcNode, void * buffer, int numBytes,
                char * msg = "<none>" );
extern void MyMPIDistribute( void * buffer, int numBytes,
                char * msg = "<none>" );

// destNode     : receiver of the gathered data
// buffer       : always points to head of entire buffer
extern void MyMPIGather( const int destNode, void * src, void * dest,
                const int numBytes, char * msg = "<none>" );

template < class C_Data >
extern void MyMPIAllgather( C_Data * src, C_Data * dest,
                const int myStart, const int myEnd, char * msg );


// The macros are relative to nodes (i.e., P_RTS)
#define RTS_NPROCS      ( P_RTS.VPNum() )
#define MYPROC          ( (int)P_RTS.VPId() )
#define FIRST           ( MYPROC == 0 )
#define LAST            ( MYPROC == ( RTS_NPROCS - 1 ) )
#define FIRST_LAST      ( FIRST || LAST )

#define THEMASTER       ( MYPROC == 0 )

// Under Aurora, the following macros are relative to 'myTeam'.
//      Here, all macros are relative to P_RTS
#define MASTER          if( ( RTSmasterOnly = true /* assign */ ) && THEMASTER )
#define END_MASTER      { RTSmasterOnly = false; /* assign */ }
#define _IPROC          ( MYPROC )
#define _NPROCS         ( RTS_NPROCS )

// Select between instrumented (or not) barriers

extern void barrier(
        char * file = NULL,             // Filename
        const int line = 0,             // Line number
        const int watchLine = -1,       // Barrier watch line
        char * name = NULL              // Named barriers
);

#define BARRIER_METHOD          barrier

#define BARRIER                                                 \
    {                                                           \
        BARRIER_METHOD( __FILE__, __LINE__, AU_BarrierWatch );  \
    }

// Support giving barriers names for easier identification
#define N_BARRIER( NAME )                                       \
    {                                                           \
        BARRIER_METHOD( __FILE__, __LINE__, AU_BarrierWatch, NAME ); \
    }

//
// run-time system : thin layer above MPI
//
class MPI_RTS
{
    private:
        int     _VPId;                  // this VP's ID
        int     _VPNext;                // next VP (round-robin)
        int     _VPNum;                 // number of VPs

```

```
    public:
        // constructor and destructor
        MPI_RTS() : _VPId( -1 ), _VPNext( -1 ), _VPNum( -1 ) { } // ctor
        ~MPI_RTS();     // dtor

        // Cache the values from MPI
        //      Not in ctor because MPI_Init may not have been called
        //      at that time.
        void QueryMPI();

        // number of virtual processors
        int VPNum() const       { return _VPNum; }

        // ID of this virtual processor
        int VPId() const        { return _VPId; }

        // next processor (round-robin)
        int VPNext()
        {
                _VPNext = (_VPNext + 1) % _VPNum;
                return( _VPNext );
        } // VPNext

        int size()              { return VPNum(); }
        int rank()              { return VPId(); }
}; // MPI_RTS

extern MPI_RTS P_RTS;

#include "stats.H"                      // For timer objects, etc.
#include "hostname.H"                   // For installPowGethostname(), etc.

extern "C" {
        // See ./chdebug.c in MPICH (mpid/ch_p4)
        extern void MPID_SetDebugFlag( int );
        extern void MPID_SetMsgDebugFlag( int );
        extern void MPID_SetSpaceDebugFlag( int );
        // Usage:  MPID_SetDebugFlag( 1 );
        // Usage:  MPID_SetMsgDebugFlag( 1 );
        // Usage:  MPID_SetSpaceDebugFlag( 1 );
};

// Send in blocks
extern void MyMPISend(
        void * srcPtr,          // What to send
        const int numBytes,     // How much to send
        const int destNode,     // Send to who
        char * msg = "<none>",
        const int myBlockSize );

extern void MyMPIRecv(
        void * destPtr,         // What to put the data
        const int numBytes,     // How much to send
        const int fromNode,     // From which node
        char * msg = "<none>",
        const int myBlockSize );

#endif // _AURORA2MPI_H_
```

Bibliography

[Abdelrahman *et al.*, 1994]

T.S. Abdelrahman, S. Brown, T. Mowry, K. Sevcik, M. Stumm, Z. Vranesic, S. Zhou, A. Elkateeb, M. Gusat, P. Pereira, B. Gamsa, R. Grindley, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, G. Ravindran, S. Srbljic, and Z. Zilic. An Overview of the NUMAchine Multiprocessor Project. *Proceedings of the Canadian Supercomputing Conference*, pages 283–295, June 1994.

[Adve and Gharachorloo, 1996]

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

[Agarwal *et al.*, 1995]

A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-J. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[Amza *et al.*, 1996]

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

[Anderson *et al.*, 1995]

T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[Bal and Haines, 1998]

H.E. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, July–September 1998.

[Bal *et al.*, 1992]

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

[Bal *et al.*, 1998]

H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[Balakrishnan *et al.*, 1998]

H. Balakrishnan, V.N. Padmanabhan, S. Seshan, M. Stemm, and R.H. Katz. TCP Behavior of a Busy Internet Server: Analysis and Improvements. *Proceedings of IEEE Infocom*, San Francisco, CA, USA, March 1998.

[BBN, 1989]

BBN Advanced Computers Inc., Cambridge, MA. *Inside the TC2000*, 1989.

[Bennett *et al.*, 1990a]

J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, May 1990.

[Bennett *et al.*, 1990b]

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

[Bershad *et al.*, 1993]

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

[Booch, 1991]

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

[Boyle *et al.*, 1987]

J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[Brooks III, 1988]

E.D. Brooks III. PCP: A Parallel Extension of C That is 99% Fat Free. Technical Report UCRL-99673, Lawrence Livermore National Laboratory, Livermore, CA, 1988.

[Bugnion *et al.*, 1996]

E. Bugnion, J.M. Anderson, T.C. Mowry, M. Rosenblum, and M.S. Lam. Compiler-Directed Page Coloring for Multiprocessors. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[Buhr *et al.*, 1992]

P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and C.R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[Buhr *et al.*, 1995]

P.A. Buhr, M. Fortier, and M.H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[Carter *et al.*, 1991]

J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Chandra *et al.*, 1994]

R. Chandra, A. Gupta, and J.L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):14–26, August 1994.

[Cole, 1989]

M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[Coplien, 1992]

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

[Doss *et al.*, 1993]

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.

[Dubois *et al.*, 1986]

M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan, June 2–5, 1986.

[Falsafi *et al.*, 1994]

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. *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[Gamma *et al.*, 1995]

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

[Geist *et al.*, 1994]

A. Geist, A. Beguelin, J.J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM (Parallel Virtual Machine): A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[Gharachorloo *et al.*, 1990]

K. Gharachorloo, D.E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Goodman, 1989]

J.R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[Grimshaw *et al.*, 1996]

A.S. Grimshaw, A. Ferrari, and E.A. West. Mentat. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[Gropp *et al.*, 1994]

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

[Gupta *et al.*, 1991]

A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, June 1991.

[Halstead, 1985]

R.H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Johnson *et al.*, 1995]

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

[Kalé and Krishnan, 1996]

L.V. Kalé and S. Krishnan. CHARM++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[Keleher *et al.*, 1992]

P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[Keleher *et al.*, 1994]

P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 Winter USENIX Conference*, pages 115–131, January 1994.

[Koelbel *et al.*, 1994]

C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, January 1994.

[Kuskin *et al.*, 1994]

J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J.L. Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Lamport, 1979]

L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[Lenoski *et al.*, 1990]

D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In Jean-Loup Baer and Larry Snyder, editors, *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, June 1990. IEEE Computer Society Press.

[Lenoski *et al.*, 1992]

D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Levelt *et al.*, 1992]

W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. A Comparison of Two Paradigms for Distributed Shared Memory. *Software—Practice and Experience*, 22(11):985–1010, November 1992.

[Li *et al.*, 1993]

X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):1079–1103, October 1993.

[Li, 1988]

K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. *Proceedings of the 1988 International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[Lu *et al.*, 1995]

H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. *Proceedings of Supercomputing '95*, December 1995.

[Lu *et al.*, 1997]

H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, June 1997.

[Lu, 1997a]

P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. *Proceedings of the 11th International Parallel Processing Symposium*, pages 467–473, Geneva, Switzerland, April 1997.

[Lu, 1997b]

P. Lu. Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library. *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 145–158, Portland, Oregon, U.S.A., June 1997.

[Lu, 1999]

P. Lu. Using Scoped Behaviour to Optimize Data Sharing Idioms. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Programming and Applications, Volume 2*, pages 113–130. Prentice Hall PTR, Upper Saddle River, New Jersey, 1999.

[Lu, 2000]

P. Lu. Implementing Scoped Behaviour for Flexible Distributed Data Sharing. *IEEE Concurrency*, 8(2), April–June 2000. in press.

[Message-Passing Interface Forum, 1997]

Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*. MPI Forum, June 1997.

[Mullender *et al.*, 1990]

S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[O'Farrell *et al.*, 1996]

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

[Parsons *et al.*, 1997]

E.W. Parsons, M. Brorsson, and K.C. Sevcik. Predicting the Performance of Distributed Virtual Shared-Memory Applications. *IBM Systems Journal*, 36(4):527–549, 1997.

[Reynders *et al.*, 1996]

J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[Rinard *et al.*, 1993]

M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6), June 1993.

[Sandhu *et al.*, 1993]

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

[Sandhu, 1995]

H.S. Sandhu. *Shared Regions: A Strategy for Efficient Cache Management in Shared-Memory Multiprocessors*. PhD thesis, Graduate Department of Computer Science, University of Toronto, July 1995.

[Saphir, 1994]

W. Saphir. Sorting Out Communication Libraries: A Comparison of NX, CMMD, PVM, and MPI. Tutorial at Supercomputing '94, November 1994.

[Scales and Lam, 1994]

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

[Schaeffer *et al.*, 1993]

J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, August 1993.

[Schoinas *et al.*, 1994]

I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[Shi and Schaeffer, 1992]

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

[Singh *et al.*, 1991]

A. Singh, J. Schaeffer, and M. Green. A Template-based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[Singh *et al.*, 1992]

J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: The Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Singh *et al.*, 1995]

J.P. Singh, J.L. Hennessy, and A. Gupta. Implications of Hierarchical N-Body Methods for Multiprocessor Architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

[Snir *et al.*, 1996]

M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.

[Stevens, 1995]

W.R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, Reading, 1995.

[Stumm and Zhou, 1990]

M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[Thekkath and Levy, 1994]

C.A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, October 4–7, 1994.

[Thi, 1992]

Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, January 1992.

[Vranesic *et al.*, 1991]

Z.G. Vranesic, M. Stumm, D.M. Lewis, and R. White. The Hector Multiprocessor. *IEEE Computer*, 24(1):72–80, January 1991.

[Wilson, 1995]

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

[Woo *et al.*, 1995]

S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[Wright and Stevens, 1995]

G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2; The Implementation*. Addison Wesley, Reading, 1995.