Parallel Computing in a World of Workstations

Andreas Polze and Miroslaw Malek
Humboldt-University of Berlin
Department of Computer Science
10099 Berlin, Germany
{apolze,malek}@informatik.hu-berlin.de

Abstract

Todays computer systems consist of powerful workstations with fast communication links. However, until now network-based parallel computers which employ interconnected workstations as processing elements are not widely used. Poor programmability of those systems as a whole seems to be one of the main reasons. New language constructs, programming paradigms and techniques for managing memory in such systems are badly needed.

In this paper we present the ‘‘Shared Objects Memory’’, an object-based distributed shared memory system. It provides a class-based programming interface for creation of shared objects of arbitrary, user-defined sizes. On top of network-transparent Mach interprocess communication, our system employs models of weak memory coherence to reduce consistency-related communication. Our approach allows for encapsulation of synchronization and communication operations in C++ classes. Thus, it provides a clean, sequential consistent view of shared objects to the programmer. ‘‘Shared Objects Memory’’ is part of our approach to network-based parallel computing within project ‘‘World of Workstations (WOW)’’.

Key words: Memory Management, Distributed Shared Memory, Object-based, C++, Mach, Memory Architectures, Software.

1. Introduction

Parallel computing on existing general-purpose hardware is a promising alternative to expensive special ‘‘supercomputers’’ when high computing power is pursued. Parallel computers are usually expensive, have special programming requirements and their hardware ages fast. This is further caused by the delay of bringing such complex products to the market. Within project ‘‘World of Workstations (WOW)’’ we develop new paradigms for programming and memory management in network-based parallel computers which integrate modern workstations connected through ATM networks. Part of the project is the ‘‘Shared Objects Memory’’-approach, an object-based distributed shared memory, which uses C++ objects as units of sharing and supports different (weak) consistency models. The current, prototypical implementation of ‘‘Shared Objects Memory’’ relies on network-transparent interprocess communication primitives offered by Carnegie Mellon’s Mach-operating system. However, we ultimately want to use a networking technology which allows to dynamically configure private channels with low latency and guaranteed communication bandwith between processing nodes. Another part of project WOW deals with ATM — our favourite candidate — and is headed to provide a timely and fault-tolerant interprocess communication facility.

a short version of this paper appeared in Proceedings of 7th IASTED/ISMM Intl. Conf. on Parallel and Distributed Computing and Systems, pp.72-75, Washington D.C., October 1995.

Network-based parallel systems differ from traditional parallel computers in several aspects. They connect a number of workstations through a fast local area network. Usually no physically shared memory is available in those systems and — due to the lack of special interconnection hardware — communication between processing elements has a much higher latency than in common parallel systems. The advent of ATM networking technology will eventually lower communication latency. However, non-uniform memory access times remain the main characteristics of network-based parallel systems and require special treatment.

2. Programming Models

Message-passing and shared-memory are two major programming paradigms which are commonly used in parallel computers. In the message-passing programming model processes exchange information by messages. But the absence of global data is inconsistent with existing sequential programming styles. Writing programs with explicit message-passing routines has shown to be error prone and tedious. In contrast, the shared-memory model provides transparent data communication. Thus it is similar to the conventional programming style and seems to be more familiar to programmers.

Distributed-shared-memory systems distribute the main memory to processors and logically share these local memories. They provide the shared-memory programming model to the user. Message-passing is internally used to support the vision of shared data. Speed of inter-node communication, data consistency model and the size of shared data chunks are main factors for remote memory access latency.

Classical distributed-shared-memory systems rely on equally sized memory pages as units of sharing. Often those pages are relatively large chunks of data. Memory pages are not related to the concept of accessing memory through variables as supported by programming languages. Thus, the problem of false sharing can occur — several logically independent variables can reside on the same memory page and are treated like a single, big shared variable. Within ‘‘Shared Objects Memory’’ the programmer can use C++ language constructs to describe size and layout of objects — the units of sharing. No false sharing of data can occur, unnecessary communication is avoided.

In our system the notion of shared objects integrates concurrency and synchronization with data abstraction. The ‘‘Shared Objects Memory’’ is implemented in C++. It consists of two parts, a (distributed) runtime system and a C++ class library. The library provides C++ base classes for shared data types and allows to express parallelism. Synchronization between tasks can be expressed by condition variables, barriers and locks. Those constructs are represented by objects of particular C++ library classes. For now, concurrent execution in our environment can be achieved either by running multiple copies of a program following the SPMD-model or by running a single multi-threaded program. In both cases tasks (or threads) can be run on different nodes of a network-based parallel computer.

Operations on shared-objects can either act on a local copy of an object or — RPC-like — work as remote invocations. Before each operation on an object’s copy the programmer has to acquire either a read- or a write-lock for the object. Then the actual operation may be performed and finally, the lock has to be released. This way data and synchronization accesses to objects can be distinguished. The memory system can implement a weakly consistent data model.

Weakly consistent data models can dramatically influence the amount of communication needed for execution of a parallel program and for keeping the illusion of a shared dataspace alive. In our system, weakly consistent models are supported by different implementations of operations *acquire* and *release* in some ‘‘Shared Objects Memory’’ base-classes.

3. Consistency Models

Maintaining a predictable view of memory across machines is called memory consistency or memory coherency. If a memory management system implements full consistency, data modified by one processor will be immediately visible to all other processors sharing the memory. Alternately, memory management may implement weaker consistency, in which updates are deferred until absolutely needed or until triggered by a special mechanism.

In order to get acceptable performance out of a multiprocessor system, data must be placed close to processors using it. Replication, together with proper allocation may significantly help for efficient data accesses in case of read sharing. This way, several copies of the data can be kept locally to several processors. However, this approach rises the *cache consistency problem*, which occurs, whenever one processor writes a replicated shared data item — the replicas must somehow be updated.

The timeliness with which cached data in a parallel program must be made consistent depends on the memory model assumed by the programmer. It has great impact on the performance of the memory management system. Programmers often assume that memory is sequentially consistent. This view matches the way memory is treated in sequential programming. It means that the ‘‘result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program’’ [Lamport 79]. In a sequentially consistent system, one processor’s update to a shared data value is reflected in every other processor’s memory before the updating processor is able to issue any other memory access.

Most parallel programs already define their own higher-level consistency requirements. This can be done by means of explicit synchronization operations such as lock acquisition and barrier entry. These synchronization operations impose an ordering on access to data within a program. Even in a sequentially consistent system, only individual memory accesses are guaranteed to execute atomically. Explicit synchronization is required for complex operations.

These observations about explicit synchronization have led to a class of weakly consistent protocols. Among them are processor consistency [Goodman et al. 89], weak consistency [Dubois et al. 86], release consistency [Gharachorloo et al. 90] and entry consistency [Bershad et al. 91]. Such protocols distinguish between normal shared data accesses and synchronization accesses. The only accesses that must execute in sequentially consistent order are those relating to synchronization. Updates to shared data in distributed memories can be handled asynchronously.

Fig. 1 illustrates how some consistency models are related to each other with respect to the level of strictness. With sequential consistency, all shared accesses must be ordered the same on all processors. Processor consistency allows writes from different processors to be observed in different orders, although writes from a single processor must be performed in the order that they have occurred. Explicit synchronization operations must be used for accesses that should be globally ordered. The primary benefit of processor consistency is that it allows a processor's reads to bypass its writes. Weak consistency treats shared data accesses separately from synchronization accesses, but requires that all previous data accesses be performed before a synchronization access is allowed. This permits load and stores between synchronization accesses to be reordered freely.

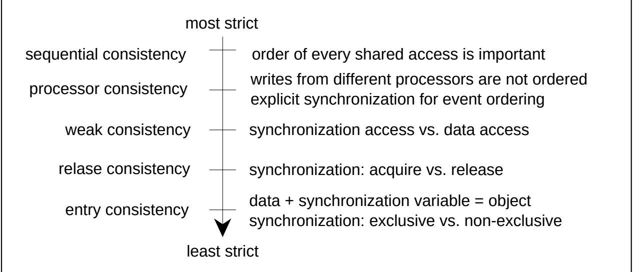
most strict
sequential consistency
order of every shared access is important
processor consistency
writes from different processors are not ordered
explicit synchronization for event ordering
weak consistency
synchronization access vs. data access
relase consistency
synchronization: acquire vs. release
entry consistency
data + synchronization variable = object
synchronization: exclusive vs. non-exclusive
least strict

Fig. 1: *Consistency Spectrum*

Release consistency and entry consistency models distinguish between data accesses and synchronization accesses. For synchronization variables, *acquire* and *release* operations can be performed. In a parallel program a critical section of data accesses is bounded by a pair of synchronization accesses.

Release consistency requires that all updates to any shared data are performed before a *release* is executed. This includes data not guarded by the just released synchronization variable — in fact, no association between shared data and synchronization variables exists. This way a process stalls at the end of a critical section until all updates to any shared data are performed.

Entry consistency associates data and synchronization variables. The only shared memory that is guaranteed to become consistent is that which can be accessed within a critical section protected by a particular synchronization variable. As a result, only relevant data is transferred. Additionally, an *acquire* operation for a synchronization variable could be pending the entire time another critical section is executed. That allows to schedule accesses to shared data independently into multiple threads on one processor. Thus communication can overlap with computation whenever a thread has to wait for a remote *acquire* to be performed.

Fig. 2 shows the behaviour of a distributed shared memory system when a processor is updating data guarded by a lock which is held in another processor's local memory. With entry consistency our processor first fetches the remote lock and the current value of a shared variable and becomes the ‘‘owner’’ of the shared variable (e.g., the lock). During the critical section all subsequent accesses to the shared variable can be handled locally. The owner of a lock can be located using a distributed queuing algorithm [Forin et al. 89] [Lee/Ramachandran 90] [Bershad et al. 91]. Finally, the lock can be released without having to communicate with remote memories.

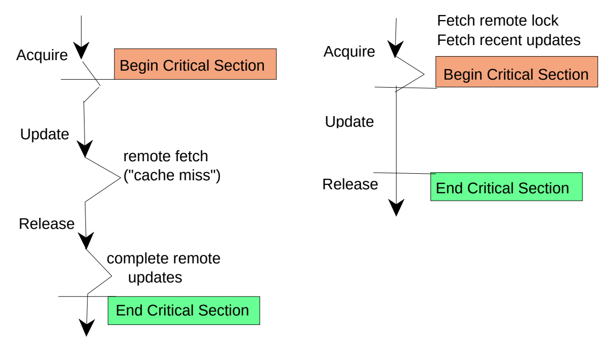

Fig. 2: *Behaviour of two weakly consistent models*

Thus, the entry consistency model is well suited for scenarios where execution of a parallel program can be divided into phases. Within a phase only threads from from one processor access a shared variable. Then, in the next phase, threads from another processor access the same variable. During each phase no consistency related communication is needed. Additionally, entry consistency distinguishes between exclusive and non-exclusive accesses to synchronization variables. So, data can be read-replicated by replicating synchronization variables as well as data and allowing exclusive accesses to a synchronization variable to be performed only by its owner.

With release consistency again a lock has to be acquired at the beginning of a critical section. However, the ‘‘owner’’ of a variable does not change by that operation. Since the release consistency model does not associate synchronization operations with particular shared data items, it is not possible to pre-fetch current values of the variables accessed in a critical section. So, eventually a ‘‘cache miss’’ will occur, extra communication is needed to fetch the most recent value of a variable. Subsequent accesses to that variable can be performed locally. Finally, the lock protecting a critical section must remain held until the shared data values have been updated in remote memories. This results in a delay when performing operation *release*.

The release consistency model is very well suited if writes to a shared variable usually occur from the processor which owns the variable (lock). Other processors are assumed to either perform write accesses seldom or not at all. If shared data and synchronization variables are tied together like with entry consistency, then even the ‘‘cache misses’’ can be avoided under the release consistency model.

In the ‘‘Shared Objects Memory’’-system we support both, release consistency and entry consistency for shared objects. An association between data and synchronization variables is made on the base of objects — each shared object is implicitly tied to a synchronization variable. This way we free the programmer from explicitly coding special function-calls to establish associations between synchronization variables and data like in the Midway system [Bershad et al. 91]. In our system, shared data is implemented by replicated objects. Two base classes exist which implement the Mach-IPC operations necessary to realize either release consistency or entry consistency for those replicated objects. A programmer who wants to have a shared data structure relying on one of the consistency models simply has to derive a class from the appropriate ‘‘Shared Objects Memory’’ base-class. We support the explicit migration (change of ownership) for objects treated under the release consistency model.

4. Overall Architecture

In Fig. 3 we show two parallel tasks running on top of the ‘‘Shared Objects Memory’’ (SOM). Both communicate with a third component: the object repository task. Such a task exists for each class of objects maintained by our system, it contains information about the locations of object’s replicas. In addition to threads belonging to a parallel program, each parallel task has a consistency manager thread. This thread is implicitly created and managed by the SOM runtime. Fig. 3 shows the flow of messages between the three Mach tasks. This communication is network-transparent, thus each task can run on a different node in a network-based parallel computer.

We will now discuss creation and maintainance of two shared objects in greater detail. In Fig. 3 both parallel tasks create a C++ object `o1`. The second tasks additionally creates an object `o2`. These objects are instances of class C which has to be a subclass of class `SOM`, the ‘‘Shared Objects Memory’’-base class. Objects in the ‘‘Shared Objects Memory’’ are referenced by unique integer identifiers. Those identifiers can be explicitly assigned by the programmer or they can be assigned by a parallel preprocessor to the standard C++ compiler. A third method exists: objects are numbered in the order of their occurence in a task (e.g., their constructors are called). This is the default behaviour if none of the above methods is used.

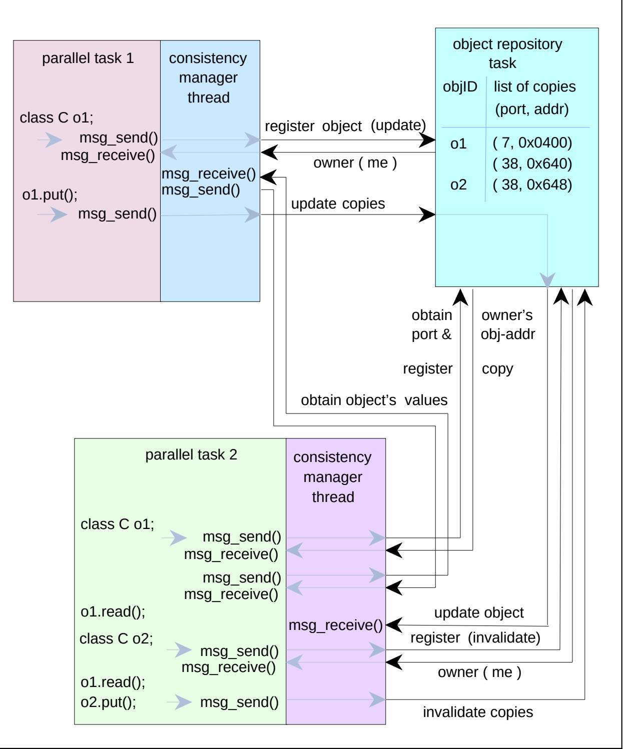

Fig. 3: *Shared Objects Memory — Architecture*

After creation of an object each task (e.g., the object’s constructor) registers the object with the object repository task. This task is accessible through the Mach `netmsg-sever` under a well-known name. The repository maintains a table of object identifiers. For each object identifier a list of references to the object’s copies is stored. Each entry within this list consists of a Mach port and an address. The port denotes a Mach task whereas the corresponding address specifies the location of an object’s copy in the particular Mach task. A task which registers a newly created object receives a reply: the port and object address of the object’s owner. In Fig. 3 task 2 figures out that there is already an owner for object `o1` registered with the repository. So it sends an additional message to obtain object `o1`’s values from the other task.

Once an object has been created and registered with the repository the `SOM` base class provides two member functions to either update or invalidate all copies of the object. Both functions result in sending messages to other tasks. Those messages are received by the peer task’s consistency manager thread. To avoid object inconsistencies due to consistency manager operations a *mutex* coordinates local accesses to `SOM`-objects.

Class `SOM` does not associate any policy with its update and invalidate operations. However, `SOM`’s update and invalidate member functions can be used to implement a particular consistency protocol. For example, class ENTRY_CONS uses those member functions to implement the entry consistency protocol. Besides class ENTRY_CONS, class RELEASE_CONS belongs to our system. It implements the release consistency protocol for shared objects. Both classes provide functions acquire_write_lock and acquire_read_lock to get either exclusive or non-exclusive access to an object. Additionally function release_lock exists in both classes.

Programmer-defined shared data structures now can be implemented by deriving subclasses from class ENTRY_CONS or class RELEASE_CONS. As an example in Fig. 4 we present a shared counter class. Replicated objects of that class are kept consistent according to the entry consistency model. Class counter has two component functions: a function show which prints a counter's value and an overloaded operator ++ which increments a counter object. Additionally the class has a constructor and two inline-functions. The functions size_of() and assign() have to be defined within each shared object's class, they are called by the *Shared Objects Memory*-runtime system.

```
class counter : public ENTRY_CONS {
        int _i;
public:
        counter( int val ) { _i = val; }
        virtual int size_of()
                  { return sizeof( *this ); }
        virtual void* assign( void* p )
                  { *this = *((counter*) p); }

        counter & operator++();
        void show();
};

counter & counter::operator++() {
        acquire_write_lock(); _i ++;
        release_lock(); return * this;
}

void counter::show() {
        acquire_read_lock();
        printf("val: %d\n", _i);
        release_lock();
}
```

Fig. 4*: Shared Class: Counter*

One should notice, that functions ++ and show provide a sequential consistent interface to instances of class counter. All the synchronization and replication operations for that class are completely hidden from the user. Here the advantage of integrating concurrency and synchronization with the notion of objects becomes quite clear: We can reduce consistency related communication by employing replication and a weak consistency scheme during implementation of class counter but are able to present an easy-to-use sequential consistent interface to the user of counter-objects.

5. Container classes for co-located objects

As demonstrated with class counter, user-defined shared objects may be implemented as instances of a class derived from ENTRY_CONS or RELEASE_CONS. A shared object is realized as a set of replicated C++ objects in different Mach-tasks. A weak consistency protocol is used to maintain a consistent view on shared data. With each of the replicas making up a shared object, a mutex-structure is associated to allow for implementation of the consistency protocol.

User-defined objects are often small, typically only a few bytes. Often those objects can be grouped into sets — like all the objects in a linked list. It seems to be inefficient to perform all the consistency-related operations for each object in such a set separately.

Traditional distributed shared memory systems are page-based. Several data items are placed on the same page by the compiler — co-locations between data items are established implicitly. Read and write access rights are maintained once for all items on a page. This is more efficient than managing each data item separately — however, it raises the problem of false sharing of non-related items.

Within our *Shared Objects Memory*-system we allow the programmer to explicitly establish co-locations between objects. Thus, all the objects of a linked list can be co-located and be treated like a single shared data. The programmer has full control over each group of co-located objects, no false sharing can occur.

With container class DYNAMIC_MEM our system provides special, overloaded versions of C++ operators new and delete. Each instance of class DYNAMIC_MEM is itself a shared object of a particular, user-defined size. This object simply provides dynamic memory for the creation of smaller, co-located objects of arbitrary classes via the overloaded operation new.

Only one mutex structure is required for each DYNAMIC_MEM object to coordinate accesses to all the objects co-located on the particular DYNAMIC_MEM object. All the co-located objects are transferred as a big data chunk whenever the current value of one of the objects has to be obtained. Thus, traversal of a linked list of co-located objects causes transmission of a single shared object over the network instead of separate transmission of each of the objects within the list.

Our container class DYNAMIC_MEM serves a similar purpose like pages in a classical distributed shared memory system. However, two main differences have to be noted: Firstly, in contrast to shared memory pages, with our approach the programmer has full control over co-location of objects on DYNAMIC_MEM objects. No false sharing of data can occur. Secondly, all the consistency-related operations on DYNAMIC_MEM objects are entirely handled in user-space. If lock acquisition for a DYNAMIC_MEM object requires transmission of that object over the network, only the thread performing the actual access stalls. Other threads in the same task can continue, this way communication overlaps with computation. On the other hand, in a classical distributed shared memory system a page fault occurs whenever an access to an out-dated shared page is attempted. This page fault blocks the whole task and the communication latency cannot be hidden.

6. Related Work

A number of distributed shared memory implementations have been described in literature, [Nitzberg/Lo 91] presents an overview.

The MUNIN system [Carter et al. 91] was one of the first software DSM systems which used release consistency as a model of memory coherence. Release consistency enables the system to merge page updates in order to propagate them in a single message on the next release operation. MUNIN offers multiple consistency protocols. Sharing annotations denote the protocol to be used with respect to a particular shared variable. Possible annotations are, for example, 'read-only', 'write-shared'. 'producer-consumer', 'migratoy' or 'conventional'. However, to exploit all those sharing strategies, most of the consistency aspects have to be controlled by software and the advantage of cheap hardware support by the MMU is partly lost. If the grainsize of shared data items tends to be small, then the page-based release consistency is probably less suitable than true sharing at the level of programmer-defined objects.

MIDWAY [Bershad et al. 91] proposes the entry consistency model of page coherence. It tries to minimize communication costs by aggressively exploiting the relationship between shared objects and the synchronization variables which protect them. However, those relationships have to be established explicitly by calls to special functions of the MIDWAY runtime systems. So entry consistency depends on the correct use of synchronization primitives throughout the whole parallel program.

The PANDA system [Assenmacher et al. 93] provides a page-based distributed shared memory together with a C++ user-level thread implementation based on a pico-kernel. PANDA employs page differencing to reduce communication bandwidth requirements when transmitting shared memory pages. It supports migration of threads and user-level objects. With PANDA the programmer can dynamically specify clusters (co-locations) of related objects which indicate to the system thata set of objects should be treated as unit of sharing and mobility. This helps to lower the probability of false sharing.

The parallel programming language ORCA [Bal/Kaashoek 92] provides sharing at the level of shared objects. ORCA introduces its own specialized programming model, this way it avoids some problems concerning compliance with existing programming languages. In a distributed prototype environment ORCA has been implemented based on object replication and reliable broadcast to achieve consistent object sharing. ORCA uses a write-update protocol to maintain consistency.

7. Conclusions

We have described the first prototype of ''Shared Objects Memory'', an object-based distributed shared memory system. It consists of a C++ class library and a Mach-based runtime support. ''Shared Objects Memory'' deals with non-uniform memory access times in distributed environments by replication of objects and by exploiting locality of reference when accessing shared objects. By supporting weakly consistent data models consistency-related communication is reduced, communication is overlapped with computation. ''Shared Objects Memory'' is part of project ''World of Workstations (WOW)'' which has the ultimate goal of developing new programming and memory management paradigms for running parallel programs in networked (ATM) environments.

The ''Shared Objects Memory'' integrates concurrency and synchronization with the notion of objects. It allows to encapsulate synchronization operations within the implementation of classes. Clients of thoses classes get access to a sequential consistent interface, no explicit synchronization operations are necessary outside the object's interface.

In contrast to common page-based memory management schemes, ''Shared Objects Memory'' gives the programmer full control of objects as units of sharing. Therefore, in our system false sharing of data is not an issue. With the example class counter we have shown how a shared class can be implemented simply by derivation from a shared object's base class. Currently, the classes ENTRY_CONS and RELEASE_CONS implement different consistency schemes for replicated objects. We have discussed the issue of co-locations of objects. By providing class DYNAMIC_MEM whose instances implement shared free store, we give the programmer full control over placing of objects on an efficiently implemented ''shared data segment''. In the future additional classes will support object migration/placement and remote invocations on a single copy of a shared object.

References

[Assenmacher et al. 93] H.Assenmacher, T.Breitbach, P.Buhler, V.Hübsch, R.Schwarz;
PANDA — Supporting Distributed Programming in C++ Proc. of 7th European Conference on Object-Oriented Programming ECOOP'93, LNCS vol. 707, 1993, pp. 361-383.

[Bershad et al. 91] B.N.Bershad and M.J.Zekauskas;
Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors; Technical Report CMU-CS-91-170, School of CS, Carnegie Mellon University, September 1991.

[Dubois et al. 86] M.Dubois, C.Scheurich, and F.Briggs;
Memory Access Buffering in Multiprocessors; in Proceedings of the 13th Annual Symposium on Computer Architecture, pp. 434-442, June 1986.

[Carter et al. 91] J.B.Carter, J.K.Bennet, and W.Zwaenepoel;
Implementation and Performance of Munin; in Proceedings of the 13th ACM Symposium on Operating System Principles, pp. 152-164, October 1991.

[Chandra at al. 92] R.Chandra, A.Gupta, J.L.Hennessy;
Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language; Technical Report CSL-TR-92-511, Computer Systems Lab. Stanford University, 1992.

[Forin et al. 89] A.Forin, J.Barrera, M.Young, and R.Rashid;
Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach; in Proceedings of Winter 1988 Usenix Conference, January 1989.

[Gharachorloo et al. 90] K.Gharachorloo, D.Lenoski, J.Laudon, P.Gibbons, A.Gupta, and J.L.Hennessy;
Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocesors; in Proceedings of the 17th Annual Symposium on Computer Architecture, pp. 15-26, May 1990.

[Goodman et al. 89] J.R.Goodman, M.K.Vernon, and P.J.Woest;
Efficient Synchronization Primitives for large-scale cache-coherent Multiprocessors; in Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 64-75, April 1989.

[Lamport 79] L.Lamport;
How to Make a Multiprocessor Computer that correctly executes Multiprocess Programs; IEEE Transactions on Computers, C-28(9):241-248, September 1979.

[Lee/Ramachandran 90] J.Lee and U.Ramachandran;
Synchronization with Multiprocessor Caches; in Proceedings of the 17th Annual Symposium on Computer Architecture, pp. 27-37, May 1990.

[Nitzberg/Lo 91] N.Nitzberg, V.Lo;
Distributed Shared Memory: A Survey of Issues and Algorithms; IEEE Computer, August 1991, pp. 52-60.