Sparks: Coherence as an Abstract Type

Pete Keleher

Department of Computer Science
University of Maryland
College Park, MD 20742
keleher@cs.umd.edu

Abstract

We are currently designing Sparks, a protocol construction library that we hope will allow us to improve the performance of DSM systems to within a few percent of tightly-coupled multiprocessors. Sparks’ abstractions will allow us to cleanly and systematically explore the design space of high-level synchronization operations, rather than proposing and implementing new operations in an ad hoc fashion. Sparks’ basic abstraction is the coherence history, an object that summarizes past coherence actions to shared segments. Our emphasis here is more on creating and investigating the abstractions that make a broad variety of optimizations possible, rather than on the individual optimizations themselves. However, we will thoroughly quantify the performance gains allowed by the synchronization types created via the Sparks library.

Our overall goal is to improve DSM performance. We will gauge our success by targeting applications from benchmark suites such as SPLASH-2, as well as representative applications from computational chemistry, biology, and satellite image analysis. Sparks’ history abstraction will be used to make several important contributions towards our performance goal: (1) efficient techniques to implement high-level synchronization, (2) efficient automatic prefetching using prefetch playbacks, and (3) external interfaces to run-time libraries and automatically parallelized code sections. By improving DSM efficiency, we hope to make the shared memory paradigm more appealing, and therefore useful, to the research community.

1 Introduction

Shared memory is a more intuitive programming model than alternatives such as message-passing. Software distributed shared memory (DSM) systems provide the abstraction of shared memory to applications running on networks of workstations and distributed memory machines such as the SP-2, CM-5, and Paragon. Unfortunately, the latencies for global operations in either environment are several orders of magnitude more expensive than on tightly-coupled multiprocessors. As a result, early DSMs performed well for only a restricted class of applications.

Previous work [5, 7] addressed part of the problem by proposing weak memory consistency models. These memory models allow processors’ views of shared memory to temporarily diverge, bringing them back into agreement only at subsequent synchronization. This work significantly broadened the class of applications that performs acceptably on DSMs, but falls short of allowing DSMs to rival the performance of multiprocessors in general.

We are currently designing Sparks, a protocol construction library that we hope will allow us to improve the performance of DSM systems to within a few percent of tightly-coupled multiprocessors.
Sparks’ basic abstraction is the coherence history. A history is an object that summarizes past coherence actions to shared segments. The Sparks history objects can be used to create high-performance synchronization types, prefetching strategies, and interfaces. Our emphasis is on creating and investigating the abstractions that make a broad variety of optimizations possible, rather than on the individual optimizations themselves. However, we will thoroughly quantify the performance gains allowed by the synchronization types created via the Sparks library.

Sparks will be implemented in the context of the Coherent Virtual Memory (CVM) system, a new DSM that has advanced support for multiple protocols, multi-threading, and prefetching. Specific concentrations of the Sparks research will include:

- **High-level synchronization** - Sparks’ abstractions will allow us to cleanly and systematically explore the design space of high-level synchronization operations, rather than proposing and implementing new operations in an ad hoc fashion. We will place special emphasis on the interaction of synchronization and protocol design.

- **Prefetch Playbacks** - Our simulation results indicate that a majority of data accesses in even fairly complicated applications can be predicted by using runtime analysis. This analysis, in combination with Sparks mechanisms, can cleanly record data transfers, associate them with individual synchronization objects, and play back the associated data transfer during subsequent iterations. We will develop a suite of prefetch heuristics that are automatically triggered by access patterns, and evaluate their performance in the context of CVM.

- **External interfaces** - Combining DSM support with run-time libraries and automatically parallelized code sections has enormous potential. One of the primary obstacles to reaching this potential is in the difficulty of creating appropriate mechanisms for the DSM and library code to “hand off” responsibility for data segments. In particular, the interface must allow detailed description of the current state and distribution of shared data when control is transferred. History objects can be used to create flexible and fine-grained descriptions of shared state, making clean external interfaces relatively simple to implement.

The rest of this research plan is organized as follows. Section 2 gives a brief overview of current DSM research. Section 3 gives an overview of the concepts, functionality, and projected use of the Sparks class library. Finally, Section 4 summarizes the research issues and discusses our conclusions.

## 2 Background

Distributed Shared Memory (DSM) systems support the abstraction of shared memory for applications running on loosely-coupled distributed systems, i.e. workstations on a general-purpose network (Figure 1). The DSM layer traps page faults and satisfies them by fetching data across the network.

While early systems strictly emulated the sequentially consistent [13] programming model of tightly-coupled multiprocessors, most recent systems support relaxed consistency models that produce identical results for most applications, but allow the use of many performance optimizations.

As part of my dissertation research, I defined lazy release consistency (LRC) [7], a close relation to the eager release consistency (ERC) [5] memory model. DSMs that implement ERC delay propagating modifications of shared data until they execute a release (see Figure 2), and then the modifications are performed globally. Under LRC protocols, processors further delay performing modifications remotely until subsequent acquires by other processors. Additionally, the modifications are only performed at the processor that performed the acquire (see Figure 3). The central intuition of LRC is that competing accesses to shared locations in correct programs will (almost)
Figure 1  Distributed Shared Memory

Figure 2  Eager Release Consistency

Figure 3  Lazy Release Consistency
always be separated by synchronization. Since coherence operations are deferred until synchronization is acquired, we can piggyback consistency information on the existing synchronization messages. In general, LRC performs better than ERC by eliminating consistency messages and further hiding the effects of false sharing.

I demonstrated the usefulness of LRC by implementing and testing TreadMarks, a software DSM that implements a multi-writer version of LRC. TreadMarks is proof that user-level DSMs are capable of efficiently running applications that were written for tightly-coupled multiprocessors, without requiring a different programming model or user annotations.

3 Sparks: Abstract Type Support for Coherence

DSMs typically separate synchronization support from shared address space support in order to achieve good performance [1, 3, 9, 6]. Such systems provide a limited set of synchronization primitives (locks, barriers), and expect application programmers to build sophisticated synchronization constructs in terms of them.

However, building high level synchronization objects using locks or barriers is often inappropriate, because the coherence constraints implied by the locks may be more strict than those needed by the high level object. Figures 5 and 4 show lock-based and Sparks-based queue implementations in an LRC environment. In both cases, process $P_1$ creates and inserts item $x$, $P_2$ creates and inserts item $y$, and $P_3$ retrieves item $x$. LRC systems transitively require the acquirer of a lock to see all shared updates seen by the last releaser. In the lock-based queue Figure 5, both $P_2$ and $P_3$ see all updates seen by $P_1$, and $P_3$ sees all updates seen by $P_2$. More to the point, $P_2$ invalidates its copy of the page containing $x$ and $P_3$ invalidates its copy of the pages containing both $x$ and $y$. However, $P_2$ never needs to see $x$. It merely transfers knowledge of $x$’s creation from $P_1$ to $P_3$. Similarly, $P_3$ does not need to know about $y$. Therefore, neither $P_2$’s invalidation of the page containing $x$, nor $P_3$’s invalidation of the page containing $y$ are necessary. In general, applying unnecessary coherence operations can waste bandwidth, create extra CPU overhead, and cause unnecessary page faults.

The Sparks class library can be used to build high level synchronization objects that accurately reflect the synchronization objects’ coherence semantics. Our approach is related to the causality annotations of CarlOS [11], but Sparks will provide a much richer set of mechanisms and finer control over the scope of consistency actions. Sparks will replace the top layer of CVM. Since coherence in LRC systems like CVM is driven by synchronization, it is also entirely proper to view Sparks as a toolkit with which to write DSM protocols.

In the following sections, we describe Sparks histories, and present several examples of their use. This list is by no means complete, we expect new uses to emerge as the system is built and we gain experience using it.

3.1 Histories

Coherence histories allow users to express and manipulate coherence constraints. By applying one node’s history at another node, the second node’s view of shared state is brought up to date with respect to events seen by the first.

More formally, a history is a partially ordered set of intervals [8], where an interval describes an interval of time for a single processor. Intervals contain write notices, which are generally just indications that a given page has been modified. Applying such a notice invalidates the associated page. However, a write notice may also contain the newly written data, and hence application of the write notice updates the page instead of invalidating it. Intervals represent a logical unit
of time; they have no correspondence with real time. In a distributed system, new intervals are typically started at each non-local synchronization event.

Figure 6 shows the example of Figure 4 with intervals labeled. Interval 0 of $P_1$, written as $i^0_1$, contains a single write notice for the page containing $x$. A history of $P_3$ at the time of the acquire could be written as:

$$H_3 = \{i^0_1, i^1_2, i^2_3\}$$

where intervals are ordered in a topological sort of the partial orderings (see Figure 7 imposed by program order, $(i^0_1$ precedes $i^1_1$), and synchronization order $(i^0_1$ precedes $i^2_2$).

Histories have three types of extent: a temporal extent, a segment extent, and a thread extent. The temporal extent specifies the interval of time for which events are summarized. A limited temporal extent can be used to name only those events that occurred during part of an execution, such as between two synchronizations. A temporal extent is described by using version vectors to summarize the earliest and latest included intervals of each processor in the system. The temporal extent of $H_3$ in Equation 1 could be written as:

$$\{\bot, \bot, \bot\} \{0,1,1\}$$

meaning that the history summarizes all intervals from the start of execution to $i^0_1$ on $P_1$, $i^1_2$ on $P_2$, and $i^2_3$ on $P_3$.

The segment extent names the region of shared memory that may be affected by the history’s write notices. The segment extent of $H_3$ is the set of pages that contain the variables $x$, $y$, and $z$. The primary purpose of the segment extent is to limit the scope of a history’s consistency actions to a subset of shared memory. In a page-based DSM like CVM, a segment consists of a set of pages. However, segments could also be composed of arbitrarily-shaped objects in distributed object systems such as Midway [1], CRL [6], or Emerald [2].

The thread extent names the set of threads whose write notices may be contained in the history. Usually this includes all threads in a system. For example, the thread extent of $H_3$ is $P_1$, $P_2$, and $P_3$. However, limiting the thread extent has several uses, including limiting the information passed to a global barrier by each node (each needs only to tell the barrier master about its own intervals), and integrating prefetching with thread scheduling on multi-threaded nodes.
3.2 Programming with Spaks

The initial prototype of Spaks will be written as a C++ class library. Later versions may migrate to a language-based approach as we expand the scope of the research to include compiler-based analysis of synchronization and automatic protocol verification.

A simplified definition of the History class is shown in Figure 8. This definition allows histories to be added, subtracted, and applied. The set_* routines allow the extents to be directly modified. Additionally, some protocol implementations of get_data() will return all data present locally whose creation is described by the history’s write notices. This apply_data routine can be used to update pages when the history is applied elsewhere. The register routine is used to tell Spaks to begin recording shared writes in a given history.

Adding histories $H_i$ and $H_j$ results in a new history that contains all intervals named in either $H_i$ or $H_j$. For example, the coherence operations that take place in a lock acquisition on an LRC system can be expressed by:

$$H_{acq} += H_{rel};$$
$$apply H_{acq};$$

The existence of a history detailing modifications to shared memory does not imply that any coherence operation has taken place. Consistency action only occurs when a history is applied to the local version of shared memory. In the above example, the first line merely creates a description of shared modifications seen by either the acquirer or releaser. No action is performed until the resulting history is applied in the second line. All three extents may be modified by an addition.

Histories may also be subtracted. Subtracting $H_i$ from $H_j$ limits the temporal scope of the resulting history to the interval of time seen by $H_j$ but not by $H_i$. History subtraction can be used to create a compact representation of all shared updates to the extents covered by history $H_{in}$ during a specific interval of time:

```c
History H_{save};
extern History H_{local};

void begin_record () {
    H_{save} = H_{local};
}
```
class History {
    TemporalExtent temporal;
    SegmentExtent segment;
    ThreadExtent thread;

    void register(int on, off);
    void operator += (History *);
    void operator -= (History *);
    void apply();
    UpdateData *getData();
    void apply_data(UpdateData *);
    void set_temporal(TemporalExtent *, TemporalExtent *);
    void set_segment(SegmentExtent *);
    void set_thread(ThreadExtent *);
};

Figure 8 History Class

History * end_record () {
    return Hlocal - Hsave;
}

where we assume $H_{local}$ is registered (recording is turned on). The history returned by end_record contains a complete record of the intervals that were created or learned about between the calls to begin_record and end_record. The next section presents possible uses of this type of construction.

3.3 High-Level Synchronization: Queues

As discussed above, unintended consequences can result from using constructs as powerful as Locks to build high level synchronization types. In the case of the lock-based queue in Figure 5, the unintended consequences are processor $P_2$’s invalidation of page $p_x$, and $P_3$’s invalidation of $p_y$. The only intended consequence is $P_3$’s invalidation of $p_x$.

The Sparks-based queue implementation in Figure 4 stores the history of the data producer with the object in the queue. When the data is consumed by $P_3$, $P_1$’s history is applied $P_3$.

3.4 Reductions and Mutual Exclusion

Many operations in parallel programs can be described as reductions, or operations that are associative and commutative. The semantics only require mutual exclusion between consecutive reducers. However, reductions are typically implemented using locks. Locks are stronger than necessary because their implementation updates later reducers with all coherence actions taken by prior reducers. The only coherence actions that need to be performed are those to the data modified by the reduction.

Reductions can be implemented in Sparks similarly to locks, except that a segment extent is used to limit the scope of the histories transferred between consecutive reducers. The below code presents the relevant aspects of a reduction acquisition:

(1) reduce_acquire(SegmentExtent *object) {
(2) send request for object to current owner
(3) \textit{extract history} $H_{\text{last}}$ \textit{from reply}
(4) $H_{\text{last}} \rightarrow \text{set segment (object)}$
(5) $H_{\text{last}} \rightarrow \text{apply()}$
(6) }

Line 4 reduces the scope of the coherence actions contained in $H_{\text{last}}$ to only those that affect the pages in \texttt{object}, and Line 5 applies the result.

### 3.5 Producer-Consumer Sharing

A common behavior in parallel programs is a stable pattern of data creation by one thread and consumption by another. Such behavior is usually termed \textit{producer-consumer}. We present two possible mechanisms to optimize data transfer when the user has indicated that producer-consumer interaction is possible.

Page-based DSMs have no explicit association between data and the synchronization used to guard it. However, a given program usually obeys a fairly simple mapping between the two. Our trace-driven simulation shows that 81\% of all access misses on shared data in Water [17], a relatively complicated molecular simulation, can be avoided by replaying data transfers. The access misses of Jacobi, a coarse-grained application, can be completely covered with simple analysis. Data access latencies directly account for 17\% of the runtime for Water on top of CVM on an eight-node SP-2, and indirectly account for more through synchronization delays and load imbalance [10]. Since our experience indicates that access miss latency is at least as important as synchronization latency, we expect prefetch mechanisms to provide significant performance benefits.

The above routines \texttt{begin record} and \texttt{end record} can be used to cleanly record data creation. The code below shows pseudo-code for a possible implementation:

```plaintext
SegmentExtent *object;
begin_record(H_{local});
...
History *H_{rec} = end_record(H_{local});
H_{rec} = set segment (object);
Msg *msg = new Msg(H_{rec} = get data());
msg = send ( consumer _proc id );
```

If $H_{local}$'s \texttt{SegmentExtent} covers all of shared memory, the recording calls generate two snapshots of all local updates made to shared memory between the calls. The routine \texttt{end record} returns a history containing only those changes made to shared memory between the two calls. The scope of this history is then limited to \texttt{object} by calling \texttt{set segment} with a \texttt{SegmentExtent} that covers only the shared pages that contain the object. The data corresponding to these modifications is accessed through the \texttt{get data()} method, and pushed to the expected consumer of the data. At the consumer side, the data is applied to the local view of shared memory, circumventing the invalidations and access misses that would otherwise have been required to retrieve the code from the producer.

A second mechanism is useful when the sharing pattern is not stable. The producer uses \texttt{begin record} and \texttt{end record} to delimit creation of data. At the first request for any page of the new data by another processor, the DSM transfers the entire block of newly created data by using the history's \texttt{get data()} method. Again, Sparks lets us easily capture and transfer the set of pages modified during the producer phase.
3.6 Prefetch Playbacks

Prefetch playbacks is a technique that allows us to record access misses taken during one iteration, and to play back the next update to the same data as an update during the next iteration. Section 3.5 describes a mechanism that allows a producer to update a known consumer. Prefetch playbacks build on this mechanism by allowing a producer to use past history to automatically identify the consumer.

Coherence histories are essentially a record of write faults. We can use a similar mechanism to record read faults. Routines analogous to `begin_record` and `end_record` are used to create a ReadHistory object that summarizes read misses taken between the two calls. These read histories are then matched with producers at the next global synchronization (barrier) to identify targets for updates. The following pseudo-code illustrates a use of this technique:

```c
for (i = -\infty; i < \infty; i++) {
    if (I am a producer) {
        int prod = -1;
        begin_record();
        produce(i);
        History *hist = end_record();
        if (hist > 0)
            send hist->get_data() to prod;
        prod = produce_barrier(hist);
    } else {
        begin_read_record();
        consume(i-1);
        ReadHistory *read_hist = end_read_record();
        consume_barrier(read_hist);
    }
}
```

The barrier routines append the histories to message arrival messages, and the barrier master matches producers to consumers by comparing SegmentExtents. During the next iteration, newly created data is pushed to the pid returned by `produce_barrier()` while waiting for the barrier to complete.

Recording and playing back data transfers was first used by the Mukherjee [14] in the context of a sequentially consistent DSM. Our work differs in two ways. First, our recording mechanisms will be part of the synchronization type definitions. The playbacks will be initiated by automatic heuristics, making them more reliable and easier to apply. With the exception of differentiating between producer and consumer barriers, all of the above mechanism could have been hidden inside the barrier routines. We pulled much of it outside the barrier routines for explanatory purposes. Second, our technique will be used for prefetching, not to maintain coherence. We will not violate correctness if subsequent iterations access different data.

3.7 Compiler/Runtime Library Interfaces

We will use Sparks to generate interfaces to code created by the SUIF [16] parallelizing compiler, and to the CHAOS [4] runtime library.

Our collaboration with Dr. Tseng's compiler group [16] will use communication analysis to determine when data will be needed by other processors. By combining this information with standard dataflow and dependence analysis, the compiler can initiate asynchronous data updates and overlap communication with computation.
Similar work is being pursued in collaboration with Dr. Saltz’s CHAOS [4] group. The general approach is to create mechanisms that let CHAOS assume sole responsibility for consistency in a confined region of shared space. The same mechanism is later used to turn the default consistency management back on. Sparks’ history abstraction allows us to develop efficient representations of shared state that can be used when control is transferred.

4 Conclusions

Parallel systems are clearly reaching a point where increasing affordability is making their widespread acceptance possible. However, this transition will not take place unless parallel machines are easy to program, and perform well. Current DSM systems handle the first problem, but do less well with the second.

Our research will bridge the gap between loosely-coupled and tightly-coupled systems by using the Sparks abstractions to reduce and optimize data movement in DSM systems. As large-scale systems increasingly resemble multiprocessor nodes connected by DSM, we expect our techniques to become common not only in clusters of stock workstations, but in the most powerful systems as well.

Bibliography


