Extending The Dimensions of Consistency: 
Spatial Consistency and Sequential Segments*

Technical Report #CS248-94

James Griffioen
Rajendra Yavatkar
Raphael Finkel

Department of Computer Science
University of Kentucky
Lexington, KY 40506-0027
{griff,raj,raphael}@ms.uky.edu

April 1994

Abstract

The Unify system is exploring scalable approaches for designing distributed multicomputers that support a shared memory paradigm. To achieve massive scalability, unify employs highly efficient communication protocols to support new weak consistency sharing models. In particular, Unify introduces the notion of spatial consistency and a non-standard memory type called sequential segments. The combination of out-of-order spatial consistency and sequential segments increases concurrency, reduces the need for synchronization, and allows the use of highly efficient non-atomic multicast protocols.

Our experience shows that there is a logical and intuitive connection (mapping) between sequential segments and a wide variety of parallel and distributed applications that require support for shared information. Moreover, the use of sequential segments results in simplified code and efficiency comparable to that of optimized message passing systems.

*This research was supported in part by NSF Grant No. NCR-9111323, Grant No. STI-9108764, and Grant No. CCR-9309176.
1 Introduction

Advances in computing and networking technologies have created challenging opportunities for research in high performance distributed computing. The wide-spread availability of high-performance uniprocessor and multiprocessor workstations combined with evolving gigabit network technologies allows us to envision a massively parallel distributed multicomputer consisting of thousands of workstations and high-performance multiprocessors scattered across a wide area network.

Unfortunately, this new geographic distribution of processing power, memory, and secondary storage has complicated, rather than simplified, the programming and execution environment. To make effective use of such an environment, we must develop programming abstractions that mask the distribution of resources but still make effective use of the distributed resources.

Distributed shared memory systems (DSM) solve (to some extent) the problem of visible distributed resources by unifying the distributed memory resources together into a single logical shared memory abstraction. Although distributed shared memory is a convenient abstraction from a programming viewpoint, the excessive performance costs have historically turn users to communication kernel based approaches where the distribution of resources is highly visible and left to the user to manage [Che84, MvRT+90, OCD+87, FSDL83, Ras86, FH88].

If we are to achieve scalable high-performance distributed shared memory systems we must develop new shared memory abstractions that not only mask the distribution of resources but also increase the level of concurrency, limit/reduce the amount of communication, and hide the propagation latencies typical of large-scale networks.

Unfortunately, abstractions provided by existing distributed shared memory systems often unnecessarily limit concurrency. For example, consider an iterative numerical algorithm where a node computes the values of the current iteration based on values produced by its immediate neighbors in the previous iteration. The common approach employs barriers to insure all nodes work in lock-step through each iteration. Such synchronization is overly restrictive; a single slow node can hinder the progress of all nodes. Or consider multiple nodes updating a single shared data object. In this case, false sharing or unnecessarily ordered (serialized) updates can cause the shared object to become a bottleneck, limiting scalability and performance. A scalable abstraction will attempt to avoid serialization and synchronization bottlenecks and increase concurrency.

A DSM abstraction that is to scale to a large number of hosts must also attempt to minimize the amount of communication needed to accomplish the user’s task. To this end, several systems
have proposed weaker forms of consistency which reduce the number of messages sent between hosts [DSB86, CBZ91, BZ91, AAL92, KBCC93]. These systems extract information from the synchronization events inherent in many programs to delay delivery of update messages and/or batch multiple messages together. However, no single consistency method produces the least amount of communication for all possible applications. Consequently, the DSM abstraction should support multiple forms of consistency. Although multiple forms of consistency are required, the abstraction’s interface should hide the type of consistency being used from the user as much as possible.

Finally, when communication is in fact necessary, the system must use highly efficient protocols to optimize communication. In particular, the DSM abstraction should be designed in such a way that delayed messages, out-of-order messages, and non-atomic multicast messages will not have a severe (adverse) affect on the overall performance of the system. Such errors will arise frequently in a large network. Highly-reliable protocols that correct these anomalies are undesirable because they incur substantial overhead costs even when no errors occur. To achieve maximum efficiency and scalability, the system should support low-overhead protocols that permit such errors to occur. Consequently, the sharing abstraction should allow applications that can tolerate such errors to receive the performance improvement benefits of low-overhead communication.

This paper describes the scalable sharing abstractions provided by the Unify distributed shared memory multicomputer system [GYF93]. In particular, Unify introduces the notion of spatial consistency and a non-standard memory type called sequential segments that together provide increased concurrency, reduced communication, and the use of highly-efficient low-overhead protocols.

The paper begins with a brief overview of the Unify architecture in sections 2 and 3. Sections 4 and 5 introduce the notion of spatial consistency and sequential segments. Section 6 discusses sequential segments in depth and describes our experience with their use. Section 7 describes our prototype implementation and some examples involving sequential segments.

## 2 Unify: An Overview

The Unify multicomputer consists of a large number of nodes (workstations or small multiprocessors) each executing the Unify distributed operating system. Unify integrates this large set of independent machines into a whole that resembles a single large shared memory multiprocessor. For the purposes of this paper we will focus our attention on Unify’s memory model. A complete overview of the Unify system can be found in [GYF93].
The Unify memory model consists of a globally addressable virtual address space. Unlike most conventional virtual memory approaches, Unify supports a single, large, virtual address space. The global virtual address space is shared across all Unify nodes, facilitating easy data sharing between processes regardless of their location. All computations throughout the entire multicomputer execute in this single large address space.

Figure 1 shows the memory space organization in Unify. Unify partitions the global virtual address space into segments. A *segment* is a contiguous region of the global virtual address space into which data can be stored or retrieved. Each segment is defined by a starting address, a length, and a set of attributes. Sharing is at the granularity of segments. Because segments have variable length, the granularity of sharing can be made as large or small as desired. Unify further organizes the global address space by grouping segments into clusters and workspaces.

A *cluster* defines a group of related segments that the operating system (and applications) manipulate as a single logical entity. The individual segments that comprise a cluster may have different attributes thereby allowing the individual components of a cluster to behave differently depending on their use. Clusters arise naturally in many applications. For example, consider an object-oriented application where each object contains two segments: a methods (code) segment and a data segment. Another example might be an object consisting of multiple data items, each with its own particular access pattern and consistency requirements. Clustering provides the logical grouping required to conveniently manipulate such related segments.

The *workspace* abstraction serves as the second level of grouping in the Unify memory organization. Each workspace defines the set of clusters (and thereby segments) visible to a particular computation. The basic unit of computation in Unify is called a *worker*. At most one worker inhabits each workspace. For the purposes of this paper we will use the terms worker, process, and thread interchangeable.\(^2\)

### 3 Synchronization

Synchronization in Unify sets it apart from all other DSM models. Most other models provide either locks, barriers, or both [GLL+90, CBZ91, BZ91, SR92], which severely limit performance.

\(^1\)Recent hardware advances have made very large virtual address spaces possible. Several conventional architectures now provide 64-bit address spaces capable of holding vast amounts of data [CLBHL92, GBR+92, Cor91, Cor92]. Although Unify assumes a large global address space, a small global address space (for example, a 32-bit address space) would only affect its scalability, not its functionality or performance.

\(^2\)Although workers differ from the standard notion of processes and threads, the distinction is unimportant to our discussion. [GYF93] discusses the worker abstraction in greater detail.
Figure 1: Two views of the memory model in Unify. The left view shows the logical organization, while the right view shows the arrangement of the segments in the global address space.
and scalability. All these models make the underlying assumption that synchronization accesses must be sequentially consistent. That is, all sites must observe the same set of synchronization accesses in the same order.

However, as explained later in Section 6.3, many applications do not rely on ordered synchronization access, so Unify does not require (all) synchronization accesses to be sequentially consistent. Moreover, Unify does not insist that all sites see a synchronization access at the same time. This lack of sequentially consistent synchronization accesses results in better concurrency which ultimately translates into improved job throughput.

The key to supporting such synchronization accesses is the abstraction of *eventcounts* [RK79]. The operations associated with eventcounts do not care *when* an event occurred, only *that* the event occurred. Consequently, the ordering of synchronization events is irrelevant. Unify provides a modified version of eventcounts and sequencers based on lazy evaluation. A more detailed discussion can be found in [GYF93].

4 Consistency

The consistency model in Unify is based on the observation that every distributed (parallel) application has its own application-specific consistency requirements [AAL92]. By providing a variety of consistency guarantees with varying performance costs, applications can select an appropriate guarantee with the desired cost.

Unify’s consistency model is innovative in several ways. First, Unify supports a wide range of consistency guarantees. Second, unlike existing distributed shared memory systems, Unify defines consistency of a shared segment across two dimensions (time and space) rather than just one dimension (time). We refer to consistency in the time dimension as *temporal consistency* and consistency in the space dimension as *spatial consistency*.

*Temporal consistency* is the classical way of viewing consistency. When a shared segment is modified concurrently at several sites, temporal consistency determines when (and in what order) changes to a shared segment are made visible to all the sites interested in a segment. Unify supplies a set of basic consistency management primitives that allow an application to select (on a per-segment basis) the appropriate temporal consistency semantics from a spectrum of consistency protocols. The temporal primitives set provides guarantees ranging from *sequential* consistency ([IVY Li86] and Mirage [FP89]) to weaker forms of consistency such as *release* (DASH [GLL90] and Munin
[CBZ91]), entry (Midway [BZ91]), or buffered consistency (Beehive [SR92]). In addition Unify supports a variety of automatic methods such as eventual consistency and best-effort consistency. Although weak temporal consistency can substantially improve performance, it is not central to the focus of this paper. Consequently, we refer the interested reader to [GYF93] for a detailed discussion of unify’s temporal consistency methods.

Spatial consistency defines a new unexplored dimension of cache consistency. Spatial consistency, like temporal consistency, contains both strict and weak consistency forms, with weak forms providing substantial performance improvements. In particular, spatial consistency determines the relative order of the data contained in the various replicas of a segment. For many distributed applications the relative order of the data items within a segment is unimportant, only the data items themselves are important. Examples of such applications include associative or content-addressable memory (e.g., tables, directories, or name registries), distributed log files, databases, distributed voting algorithms, and distributed/parallel reduction algorithms. The following section discusses the characteristics and implications of the spatial consistency dimension.

4.1 Spatial Consistency

Spatial consistency determines the relative order of the contents of the replicas of a segment. We say that the cached copies of a shared segment are in-order consistent if the order of the data within the segment is the same at all sites. Similarly, if all the cached copies contain the same information, but in a different order, we say the cached copies are out-of-order consistent.

For example, consider a log segment containing several log entries that is cached at three sites: A, B, and C. Suppose a worker at site A appends an entry X to the log, and at the same time a worker at site C also appends an entry Y to the log. Both A and C propagate their change to all the other sites. The replicas are in-order consistent if the log entries appear in the same relative order at all three sites (e.g., all three sites see the order as X and then Y). The replicas are out-of-order consistent if the cached copies at each site contain the same log entries but the relative order of the entries within the segment differ (for example, the entries at site A are ordered X and then Y, but at B and C the entries are ordered Y and then X).

Most shared-memory models only provide random access memory. Consequently, any consistency guarantee other than in-order consistency (e.g., out-of-order consistency) is of little, if any, value. Unify, however, supports two additional types of memory segments: associative memory and sequential memory. Given these two new types of memory, out-of-order consistency is quite
meaningful and can substantially improve performance. Section 5 discusses these two new memory
types in greater detail and describes the benefits of spatial consistency in such a system.

5 Segment Types

To obtain the most benefit from spatial consistency, Unify introduces two new types of memory
regions (segments): **associative segments** and **sequential segments**. By default, segments provide
random (direct) access to the data elements stored in the segment (we call these **random access
segments**). However, sequential and associative segments define a new set of operations that are
allowed on segments. As the names imply, associative segments provide keyed access while sequen-
tial segments provide front-read, end-write access. By supporting multiple segment types, Unify
allows applications to choose the most appropriate storage mechanism for the task.

Associative segments are organized as a collection of <key, value> pairs and only allow asso-
ciative access (i.e., sometimes called content-addressable). Read and write operations for such
segments require a key as an argument. Clearly, no order is imposed on the contents of the seg-
ment, allowing the use of out-of-order consistency. Associative segments are useful in a wide variety
of applications such as distributed databases, distributed directory services, or name servers that
support associative lookup on shared repositories of data.

A sequential segment is a segment that is intended to be accessed in a sequential manner. When
a segment of this type is included in a workspace, Unify associates a next_read pointer with the
segment (on a per-workspace basis) to keep track of the next location within the segment to be
accessed via a read operation. Thus, read(SeqSeg) returns the contents of the location at the
current workspace’s next_read address. write(SeqSeg, value) writes the given value to the last
location (next_write) in the segment. Unlike Next_read, Next_write is effectively a global pointer
shared by all workspaces.

As mentioned above, the relative ordering of values in an associative memory is of no interest
and no use to the application. Moreover, the relative ordering of the values in a sequential memory
segment (e.g., a log file segment) often turns out to be irrelevant. For example, a shared log file
might record timestamped error messages from several printer daemons. Because the data itself
contains ordering information (i.e., timestamps), there is no need to expend extra effort to insure
that log entries appear in the correct chronological order across all cached copies. Applications that
tolerate (or are indifferent to) out-of-order consistent segments can achieve substantially improved
performance resulting from the reduce synchronization and communication overhead of out-of-order segments.

Both associative and sequential segments also support a user-specifiable temporal consistency component allowing a user to chose form a wide range of temporal/spatial combinations.

The following section describes the specific operations available on sequential segments and their semantics. It also discusses the implications of the use of sequential segments in the development of various distribute application classes.

6 Sequential Segments

When an application creates a sequential segment, the application must specify two consistency-attributes for the segment corresponding to the temporal consistency desired and the spatial consistency desired. In addition, sequential segments, like random segments, are of fixed size. Consequently, the application must also specify the size of the sequential segment at creation time, which remains fixed over the lifetime of the segment. Finally, the application must specify a variety of optional attributes such as the segment's protection and whether is it to be shared or not.

In the following, we describe the semantics of operations performed on sequential segments and the implications of spatial consistency on sequential segments.

6.1 Sequential Segment Operations

Applications access the contents of a sequential segment via two simple operations: read and write. Read and write operations on a sequential segment are performed in only a sequential manner and thus do not require an address as a parameter. Logically, unify associates two pointers, next_read and next_write with a sequential segment. These pointers maintain the location of the next read and next write operation respectively. After execution of a read or write operation the appropriate pointer is automatically incremented. Next_write always points to the last item written in the segment.

Write(SeqSeg, value) is a non-blocking operation that appends the specified value to the end of the segment (i.e., by writing to the last location (next_write) in the segment). When the segment is shared, next_write is effectively a global pointer shared by all workspaces. The spatial consistency attribute (described below) determines the relative order of values written by different workspaces in the case of concurrent writes to a shared sequential segment.
When a sequential segment is included in a workspace, Unify associates a `next_read` pointer with the segment to keep track of the next location within the segment to be accessed in a read operation on a per-workspace basis. Thus, `read(SeqSeg)` returns the contents of the location at the current workspace's `next_read` address. If no data item has been written at that location yet, the read operation blocks until a data item becomes available. An application may reset the `next_read` pointer at any time which returns the `next_pointer` to the beginning of the segment.

Unify assumes a very large virtual space, and thus sequential segments are typically much larger than needed. However, because sequential segments are of fixed length, the possibility exists that an application may issue enough write operations to exceed the capacity of the segment (e.g., applications that assume an infinite size log file). To handle this case, Unify automatically wraps the `next_write` pointer to the beginning of the segment when the `next_write` pointer reaches the end of a sequential segment. A `wrap_around_count`, associated with each sequential segment, keeps track of the number of times the `next_write` pointer has wrapped around. The `wrap_around_count` is accessible to applications should they need that information.

Usually, the `next_read` pointer in a workspace lags behind the global `next_write` pointer. However, a wrap around can cause problems if the `next_write` pointer wraps around and advances beyond the `next_read` pointer in a workspace, overwriting the previously unread values. When a worker with an overwritten `next_read` pointer issues a `read` operation, `read` will return an error value. This allows the application to detect stale pointers and take corrective action. Should the application wish to continue, the `next_read` pointer is reset to the oldest unread value and operation continues as normal from that point forward.

Operations for creating, destroying, and sharing sequential segments also exist but are not important to this discussion.

### 6.2 Sequential Segments and Spatial Consistency

Unlike random segments, sequential segments can benefit substantially from the new spatial consistency dimension resulting in less synchronization and higher levels of concurrent access to shared segments. However, it should be noted that, irrespective of the selection of a particular spatial consistency, sequential segments have inherent semantics that are attractive for massively parallel applications. First, because the read operation simply waits for the next value available in sequence, there is no need for explicit synchronization between a reader and a writer. Instead, an update resulting from a write operation simply needs to be propagated to all the copies of the segment.
Second, sequential segments can lessen the impact large network latencies have on shared accesses. Because updates resulting from a series of write operations can be eagerly propagated to all the copies of the segment without waiting for readers to consume previously generated values, the values are immediately available when they are needed. Third, sequential segments also result in increased concurrency because a write and associated readers can proceed concurrently at different speeds.

If \textit{in-order} spatial consistency is specified, Unify ensures that all the workers see an identical ordering among all the values in a shared sequential segment. In other words, a sequence of read operations returns the same values in the same order at all workspaces. When multiple writers are involved, Unify randomly chooses a global ordering of the concurrent writes and the writes appear in that particular order at all workspaces. Thus, application has no control over the way concurrent writes are ordered, but can be assured the write will be ordered the same everywhere. If a specific ordering is required, applications can use explicit synchronization to ensure a particular order among concurrent writes.

A sequential segment with \textit{out-of-order} spatial consistency is much less demanding. No global ordering is necessary among concurrent writes to the same segment as long as all workspaces see all the values written. Thus, a sequence of read operations in one workspace may return values in a different order than in another workspace. If an application’s semantics requires that all workers see the result of a particular write operation, such constraints can be enforced using appropriate temporal consistency attribute.

There are several advantages of out-of-order consistency over in-order consistency with sequential segments. First, concurrent writes to the same segment do not require synchronization or a consistency maintenance algorithm because multiple writers can propagate updates independently. As a result, there is no synchronization overhead (i.e., synchronization messages which can be costly in a network with large latencies). Second, concurrency is increased because writers are not serialized but rather propagate updates concurrently to all other copies. Not only can the updates arrive in any order at any workspace, but any given update may reach different workspaces at different times within the constraints of the temporal consistency specification which is important in a large system with varying latencies.
6.3 Experience with Sequential Segments

We have found sequential segments extremely useful for many distributed and parallel applications both in terms of efficiency and in terms of being a suitable programming abstraction. Our experience shows that sequential segments can be used to improve the performance of applications taken from a wide range of application domains. In the following, we give a sampling of some widely used distributed application domains and illustrate how such application domains would be mapped to the sequential segment abstraction in unify.

6.3.1 Distributed Logs

Sequential segments are useful for applications such as distributed logs that are accessed in sequential manner. The log is written to frequently by multiple writers and read by multiple readers. In many cases, the ordering of entries within the log is irrelevant (e.g., if the entries contain timestamps as in a print daemon log). In this case the distributed log can be implemented very simply as a sequential segment with out-of-order spatial consistency. This provides workers at each site with an up-to-date segment view that contains all the log entries even if the same log entries may appear in different order on different sites.

6.3.2 Convergent Databases/Registries

Convergent distributed database systems and registration services allow inconsistencies between the cached copies of the data as long as the data is guaranteed to converge to a single consistent view in the absence of new updates. Information management systems based on this model of computing, such as the Grapevine registration service [BLNS82], are able to detect and adjust for inconsistent data. This style of information management has a wide variety of applications [Plu82, Moc87, BLNS82] and can be achieved efficiently and simply using sequential segments. The lack of temporal consistency constraints and tolerance of inconsistencies lends itself to a weak temporal sharing paradigm in which updates are distributed periodically. Moreover, the lack of support for atomic transactions implies that the ordering of updates from multiple sources is unimportant. Ordering of updates from the same host can be maintained via a host-specific timestamp attached to each transaction. Consequently, out-of-order sequential segments serve as a highly efficient building block on which to implement the transaction segments of convergent database systems.
6.3.3 Iterative Algorithms

A large class of scientific applications use iterative algorithms. An example of such an iterative algorithm is Successive Over Relaxation (SOR) which might be used to solve a Laplace heat equations and determine the temperature gradient on a square area given the temperature values at the area boundaries. Another example can be found in image processing where images are analyzed or processed via an iterative process on the pixel values.

In such applications, the solution is iteratively achieved by refining or processing information from the previous iteration. When the problem is parallelized, each worker computes a small portion of the result. At each step, the work typically uses values that it produced in the previous iteration as well as values produced by “neighboring” workers in the previous iteration. Consequently, at the end of each iteration, neighboring workers must exchange values.

Such algorithms can be efficiently implemented using sequential segments in Unify with minimal communication and implicit synchronization. In the Unify implementation, each portion of the problem assigned to a worker is further subdivided into multiple segments. One segment is not shared with any other workers and does not cause and sharing overhead. The other (shared) segments are declared to be sequential segments (append-only) with application-aided temporal consistency and out-of-order spatial consistency. Each segment is written by a single worker and read by at most N neighboring workers. Workers that need neighboring values use the primitive Need_updates to join the copy set of the appropriate neighboring segments.

In each iteration, a worker reads from each neighboring segment using the sequential segment read primitive (which blocks the worker until the next item is available) and then writes (appends) new values to its shared segments. Each entry written may contain ordering information by including iteration number along with the value written. No explicit synchronization is needed, and a worker can get ahead of others as long as the values it needs are available. Moreover, updates to shared segments may appear out-of-order across the copies because each entry contains ordering information. Consequently, the operating system need not expend any effort to insure in-order consistency.

6.3.4 Producer-Consumer Paradigm

Sequential segments also provide a useful abstraction for parallel applications involving a producer-consumer relationship.
In the case of a single producer/single consumer problem, the producer repeatedly produces new values and places them in a shared buffer from which the consumer retrieves them. In most distributed shared memory systems, explicit synchronization is necessary to block the producer when the buffer is full and to block the consumer when the buffer is empty. In Unify, a sequential segment can be used to implement a shared buffer. The producer simply appends the new items to the sequential segment and the consumer uses the read operation (possibly blocking) to read the new values. No explicit synchronization is necessary to block the consumer when new values are not available. If necessary, an eventcount may be used to let the producer synchronize with the consumer if the sequential segment should become full.

In applications involving M producers and a single consumer, a sequential segment is again an appropriate choice as the shared buffer. If the order in which the consumer reads the produced values does not matter, out-of-order spatial consistency can be used for the shared buffer incurring less operating system overhead and improved concurrency. Again, an eventcount can be used to block the producers if the buffer becomes full.

Sequential segments also offer a useful programming paradigm for applications involving multicast in which a single producer sends each new value to N consumers (i.e., single producer/multiple consumer). An example of such an application is distributed process control where a sensor periodically multicasts measurement samples to a group of controllers. Again, a sequential segment (with in-order or out-of-order spatial consistency depending on the desired ordering) can serve as a shared buffer between the producer and consumers. Each consumer can consume values at its own pace with read operations blocking it until a new value becomes available. Because the process control application may discard old values after some time, the producer can simply continue producing values at a fixed rate wrapping around when the segment is full with no need for explicit waiting for a slow consumer to read old values.

6.3.5 Work Heap Paradigm

A distributed implementation of a branch-and-bound or backtracking algorithm involves recursively dividing the problem into subproblems, allocating the subproblems to available workers, and evaluating the partial results to spawn new subproblems until an acceptable solution is found. Under the work heap or data pool concept, such algorithms are implemented by creating a shared heap of work. The heap keeps track of unsolved problems starting with the original problem. Every time a worker needs work, it removes a piece of work (subproblem) from the heap, divides the problem
into subproblems, starts working on one subproblem, and returns the rest of the subproblems to the heap. Workers independently and concurrently work on the subproblems. When a worker finishes a piece of work, it may generate another subproblem based on the result and add it to the heap. Such an application maps nicely onto an in-order sequential segment which represents the work heap. When new work is generated it is simply appended to the heap of work. Using explicit synchronization, available workers can then quickly retrieve work from the sequential segment.

Alternatively, many distributed applications use a client-server paradigm, in which a server provides a service to multiple clients. Client requests are issued by simply sending the request to the server which will eventually process the request and return the result to the client. In this case a heap of requests accumulates at the server and where ordering is not important. Consequently, clients can place their requests in a shared sequential segment with out-of-order consistency that is used as the work heap for the server.

6.3.6 Distributed Consensus Algorithms

Many distributed applications make use of distributed consensus or distributed voting algorithms. These algorithms might be use to agree on a particular version of a file to use or to select a new leader or master node. Often these algorithms simply involve each node placing a vote and then tallying the votes to determine the outcome (decision). In this case, an out-of-order sequential segment provides an efficient area in which to record the votes of the various processes. Each process can then independently tally the votes and obtain the outcome.

7 Experimental Results

To illustrate the advantages of using sequential segments in parallel and distributed applications, we present two case studies. The first case study involves a parallel implementation of the Jacobi algorithm to solve a Laplace heat equation and determine the temperature gradient on a square area given the temperature values at the area boundaries. The second case study involves an implementation of a distributed consensus algorithm.

7.1 Jacobi Algorithm Example

The goal of the Jacobi algorithm is to determine the temperature gradient on a square area given the temperature values at the area boundaries. The square area is discretized into grid points
Figure 2: An NxN matrix with M=9 workers is shown in the left part of the figure. Each worker computes values for its own section in each iteration. The section for each worker consists of nine segments. The figure on the right shows the sharing pattern for the Worker 5. The central segment (the white box) is used only locally. The dark grey boxes represent segments that the worker must read from in each iteration. The light grey boxes represent shared segments that the worker updates and shares with its immediate neighbors.

according to some desired granularity. A matrix of values represents the temperature at each grid point. Given the temperature matrix, the Jacobi algorithm iteratively computes the temperature values at each grid point. During each iteration, each matrix element is updated by computing the average of the values of its four nearest neighbors.

Given a large matrix (NxN) and M workers, the matrix is subdivided into M sections. Each worker is responsible for computing the values in its section as shown in Figure 2. At the end of each iteration adjacent workers exchange computed values at section boundaries, introducing the need for a synchronization mechanism.

In the following, we first present a conventional implementation of the application using barriers for synchronization and then an implementation using Unify primitives.

7.1.1 Conventional Implementation

To avoid overwriting the value of an element before it is used (by one of the neighbors), workers use a “shadow” copy of the matrix to update values while leaving the “original” matrix unchanged. Each worker independently computes the new values for elements in its section of the “shadow” copy using the old values in the “original” matrix. The workers then synchronize by waiting on a barrier. After synchronizing, each worker switches its notion of the “original” and the “shadow” copy and repeats the algorithm until the solution converges.
This implementation does not allow maximum parallelism and does also not scale well for several reasons. First, conventional method requires all workers to proceed in lock step. However, a worker only shares values with its immediate neighbors, and thus does not need to wait for other (non-adjacent) workers to compute their values before proceedings with the next iteration. Second, all the workers synchronize at a single barrier, which requires communication among all threads (unnecessary in this case) and poses a serious bottleneck as the size of the multicomputer scales up.

7.1.2 Unify Implementation

In the Unify implementation, each section of the matrix assigned to a worker is further subdivided into nine segments as shown in Figure 2. The central portion of each section is not shared with any worker and can be treated as an ordinary segment with no overhead of sharing. The other eight (shared) segments are declared to be sequential segments (append-only) with application-aided temporal consistency and out-of-order spatial consistency. Each segment is written by one worker (say, worker 5 in Figure 2) and read by at most three workers. Workers that need adjacent values use the primitive need_updates to join the copy set of the adjacent segment.

Figure 3 shows an outline of the sequence of steps executed by each worker. In each iteration, a worker reads from each adjacent segment using the read primitive used with sequential segments (blocks the worker until the next item is available) and then writes (appends) new values to its shared segments. No explicit synchronization is needed, and a worker can get ahead of others as long as the values it needs are available. Moreover, updates to shared segments may appear out-of-order across the copies because each entry contains ordering information. Consequently, the operating system need not expend any effort to ensure in-order consistency.

This implementation clearly offers more parallelism (no single point of synchronization) and scales well as the updates to shared segments only propagate to a few sites.

7.2 Distributed Consensus Algorithm

Many distributed systems involving replication or distributed repositories of data encounter the distributed consensus problem where a group of workers must reach a consensus before changing shared data or state of the system. An example is a majority voting algorithm proposed by Gifford [Gif79] used in a replicated file system. The file system is composed of a number of representatives, each
/* Code for worker I -- manages section I of the matrix */

/* Section I, subdivided into subsections Sij, j=1, .... 9 */
begin

/* first join the copy set of adjacent segments */
for (each adjacent segment Akl) do
need_updates(Ok); 

/* main algorithm */
while ( NOT Converged ) do
begin
for (subsect=1 to 9) do
begin
/* read new values from neighboring segments */
read( Adjacent_Segs(subsect) );

NewVals = compute_new_value();
write(subsect, NewVals);
end
end

Figure 3: An outline of code executed by a worker I that manages Section I of the matrix representing grid points.
containing a copy of the file system. Each representative has a number of votes and a majority is a subset of representatives whose votes sum to more than half of the total number of votes assigned.

To reach a consensus on the current version number, representatives vote and the version number proposed by a majority wins. First, consider the implementation of such a voting algorithm under a typical DSM system. In such a system, the algorithm would use a shared data structure to collect votes. Each representative must first lock (and possibly invalidate) copies of the shared data structure, update it (possibly propagating the update all other sites), and then release the lock. Such an implementation requires considerable communication overhead and severely restricts concurrency as only a single representative can update the shared data at a time.

In Unify, a shared sequential segment with out-of-order consistency provides a convenient programming abstraction for such an algorithm. Each representative simply appends its vote to the shared segment and then sequentially reads (using the sequential read primitive) enough items to determine the majority decision. Each representative can propagate its update to all others immediately and then read items from its copy independently without need for explicit synchronization. Thus, this approach provides considerable ease in programming, maximizes concurrency, and minimizes the amount of communication.

8 Implementation

We have implemented a run-time library that emulates the Unify kernel on a network of Sun workstations. Figure 4 shows the organization of the Unify implementation.

At the lowest layer, the Unify kernel provides both reliable and unreliable message passing mechanisms that allow communication among various Unify sites. The kernel provides a segment layer that implements functions such as segment creation, management of segment tables, and mapping cached copies of segment onto the underlying virtual memory hardware. The most notable feature of the implementation is that the kernel needs to provide only three primitives to support sequential segments: append, get_limit, and wakeup_beyond. Write operations on a sequential segment are translated into append operations where an append simply adds a data item to the end of the segment. The Unify kernel maintains a write_next pointer for each sequential segment and the pointer determines the location of the next append operation. When a worker reads from a sequential segment, it can query the location of the current write_next pointer by issuing the

\[^3\text{We are currently implementing and testing several of the applications described above.}\]
Figure 4: Organization of Unify implementation.

The `get_limit(SegID)` primitive that returns a pointer to the last item in the segment. The primitive is useful when a worker needs to determine the number of items in the segment. If there are currently no more items available, `wakeup_beyond` can be invoked to suspend the process until a new item is added to the segment.

We have found sequential segments particularly useful as the basis for implementing eventcounts in Unify run-time library. The Unify library implements an eventcount using an out-of-order sequential segment. An `Advance(E)` on an eventcount `E` simply appends an item to the segment. The eventcount primitive `Await(E, Value)` blocks the caller until the value of `E` has reached `Value`. Thus, the implementation of Await requires the library to block the caller until the segment has been written to `Value` times. The library routine can thus invoke `get_limit` to determine whether the eventcount has reached the specified value. If not, `wakeup\_beyond(SegID, Value)` can be invoked to block the caller until `Value` writes have been issued.

## 9 Related Work

Several existing distributed shared memory systems have experimented with weaker forms of consistency than strict consistency[Li86]. Examples include Mirage [FP89] (Δ consistency) Munin [CBZ91] (release consistency) Midway [BZ91] (entry consistency) and Clouds [RK88, DJA88] (application specifiable). Unify has benefited from these systems and provides flexible consistency protocols including these weakly consistent protocols. However, these systems have only focused
on temporal consistency methods. Unify provides new dimension to consistency via the use of
spatial consistency allowing applications not requiring inorder consistency to obtain additional
performance improvements.

The VDOM system [FL92] built on Amber [JAL+89] exhibits some of the features of unify’s
sequential segment type. However, VDOM does not contain any notion of spatial consistency.
VDOM provides high-level versioned memory objects which allow applications to randomly access
different versions of a particular objects. Like sequential segments, VDOM objects can be used
to improve the performance of iterative style applications which repeatedly create new versions of
an object. VDOM can, in fact, be build simply and efficiently on top of sequential segments and
would provide a very convenient high-level object-oriented interface to sequential segments.

10 Summary

We have described the design of the Unify multicomputer operating system and discussed how
the consistency and synchronization mechanisms in Unify combine to offer scalable solutions to
the problem of building a large scale, shared memory distributed computer. The principal ideas
in Unify are to provide different types of segment types, a wide range of of consistency models,
and efficient synchronization primitives that together make it possible to realize a wide range of
problem-specific semantics. Furthermore, Unify provides applications knobs to carefully tune the
performance depending on the scale and size of the parallelism desired. We are building a prototype
implementation of Unify and expect to have interesting results in early Spring that will be included
in the final version of this paper.

11 Acknowledgments

We would like to acknowledge Chris Diaz who worked on the user-level library emulation and Jyothi
Rajagopal for preliminary versions of various numerical applications.

References

for High Performance Distributed Shared Memory. In Proceedings of the 3rd Symposium on


