Shared Memory And Distributed Shared Memory Systems: A Survey

Krishna Kavi, Hyong-Shik Kim, University of Alabama in Huntsville
Ben Lee, Oregon State University
Ali Hurson, Penn State University

Introduction

Parallel and distributed processing did not lose their allure since their inception in 1960's; the allure is in terms of their ability to meet a wide range of price and performance. However, in many cases these advantages were not realized due to longer design times, limited scalability, lack of OS and programming support, and the ever increasing performance/cost ratio of uniprocessors [Bell 99]. Historically, parallel processing systems were classified as SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instructions Multiple Data). SIMD systems involved the use of a single control processor and a number of arithmetic processors. The array of arithmetic units executed identical instruction streams, but on different data items, in a lock-step fashion under the control of the control unit. Such systems were deemed to be ideal for data-parallel applications. Their appeal waned quickly since very few applications could garner the performance to justify their cost. Multicomputers or multiprocessors fall under the MIMD classification, relying on independent processors. They were able to address a broader range of applications than SIMD systems. One of the earliest MIMD system was the C.mmp built at CMU that included 16 modified PDP-11/20 processors connected to 16 memory modules via a crossbar switch. This can be viewed as a Symmetric Multiprocessor (SMP) or a shared memory system. The next version of a multiprocessor system at CMU was known as Cm* and can be deemed as the first hardware implemented distributed shared memory system. It consisted of an hierarchy of processing nodes; LSI-11 processors comprising clusters where processors of a single cluster were connected by a bus and clusters were connected by inter-cluster connections using specialized controllers to handle accesses to remote memory.

The next wave of multiprocessors relied on distributed memory, where processing nodes have access only to their local memory, and access to remote data was accomplished by request and reply messages. Numerous designs on how to interconnect the processing nodes and memory modules were published in the literature. Examples of such message-based systems included Intel Paragon, N-Cube, IBM' SP systems. As compared to shared memory systems, distributed memory (or message passing) systems can accommodate larger number of computing nodes. This scalability was expected to increase the utilization of message-passing architectures.

The Changing Nature Of Parallel Processing

Although parallel processing systems, particularly those based on message-passing (or distributed memory) model, have been researched for decades leading to the implementation of several large scale computing systems and specialized

supercomputers, their use has been limited for very specialized applications. One of the major reason for this is that most programmers find message passing very hard to do - especially when they want to maintain a sequential version of program (during development and debugging stages) as well as the message passing version. Programmers often have to approach the two versions completely independently. They in general feel more comfortable in viewing the data in a common global memory, hence programming on a shared memory multiprocessor system (or SMP) is considered easier. In a shared memory paradigm, all processes (or threads of computation) share the same logical address space and access directly any part of the data structure in a parallel computation. A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning, migration and load balancing. The shared memory also improves the ability of parallelizing compilers, standard operating systems, resource management and incremental performance tuning of applications.

The trend even in commercial parallel processing applications has been leaning towards the use of small clusters of SMP systems, often interconnected to address the needs of complex problems requiring the use of large number of processing nodes. Even when working with a networked resources, programmers are relying on messaging standards such as MPI (and PVM) or relying on systems software to automatically generate message passing code from user defined shared memory programs. The reliance on software support to provide a shared memory programming model (i.e., distributed shared memory systems, DSMs) can be viewed as a logical evolution in parallel processing. Distributed Shared Memory (DSM) systems aim to unify parallel processing systems that rely on message passing with the shared memory systems. The use of distributed memory systems as (logically) shared memory systems addresses the major limitation of SMP's; namely scalability.

The growing interest in multithreading programming and the availability of systems supporting multithreading (Pthreads, NT-Threads, Linux Threads, Java) further emphasizes the trend towards shared memory programming model. Finally, the formation of OpenMP group and the specification of OpenMP Fortran in October 1997, can only be viewed as a trend that drives the final nail into the coffin of message passing paradigm. The new standard is designed for computers that use either the Unix or Windows NT operating systems and employ multiple microprocessors for parallel computing. OpenMP Fortran is designed for the development of portable parallel programs on shared memory parallel computer systems. One effect of the OpenMP standard will be to increase the shift of complex scientific and engineering software development from the supercomputer world to high-end desktop work stations.

Programming Example

In order to appreciate the differences between the shared memory and message passing paradigms, consider the following code segments to compute inner products. The first program was written using Pthreads (on shared memory), while the second using MPI (for message passing systems). Both are assume a master process and multiple worker processes, and each worker process is allocated equal amounts of work by the master.

There are two major differences in the two implementation, related to how the work is distributed and how the worker processes access the needed data. The Pthread version shows that each worker process is given the address of the data they need for their work. In the MPI version, the actual data is sent to the worker processors. The worker processes of the Pthread version access the needed data directly as if the data is local. It can also be seen that the worker processors directly accumulate their partial results in a single global variable (using mutual exclusion). The worker processes of the MPI program are supplied the actual data via messages, and they send their partial results back to the master for the purpose of accumulation.

In addition to the aforementioned differences, there is another important difference. In the Pthread implementation, the user is not concerned with the data distribution, while in the MPI version, the programmer must explicitly specify where the data should be sent.

Code for shared memory using Pthreads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
double sum = 0.0;

typedef struct {        /* data structure used for work distribution */
   int n;               /* number of elements */
   double *x;           /* address of the first element of the 1st array */
   double *y;           /* address of the first element of the 2nd array */
} arg_t;

static void *worker(arg_t *arg)        /* worker process begins here */
{
   int i;
   int n = arg->n;
   double *x = arg->x;                 /* takes data location */
   double *y = arg->y;                 /* takes data location */
   double partial = 0.0;

   for (i = 0; i < n; ++i)             /* calculation */
      partial += x[i]*y[i];

   pthread_mutex_lock(&lock);                /* lock */
   sum += partial;                     /* accumulate the partial sums */
   pthread_mutex_unlock(&lock);        /* unlock */

   return NULL;
}

main(int argc, char *argv[])
{
   double x[N_VEC], y[N_VEC];               /* input vectors */
```

```
   pthread_t thread[N_PROC];
   arg_t arg[N_PROC];
   int i, status;

   generate(x, y);                          /* generates two vectors */

   for (i = 0; i < N_PROC; i++) {       /* master process distributes the work */
         arg[i].n = SZ_WORK;
         arg[i].x = x + i*SZ_WORK;
         arg[i].y = y + i*SZ_WORK;
         status = pthread_create(&thread[i], NULL, (void*) worker, &arg[i]);
         if (status) exit(1);
   }

   for (i = 0; i < N_PROC; i++) {       /* wait for all worker processes to complete */
         status = pthread_join(thread[i], NULL);
         if (status) exit(1);
   }

   printf("Inner product is %f\n", sum);
```

Code of message passing systems using MPI

```
#defineMASTER       0
#defineFROM_MASTER  1
#define TO_MASTER    2

double sum = 0.0;

static double worker(int n, double *x, double *y)
{                                           /* called by worker process */
   double partial = 0.0;
   int i;

   for (i = 0; i < n; ++i)                  /* calculation */
         partial += x[i]*y[i];

   return partial;
}

main(int argc, char *argv[])
{
   double x[N_VEC], y[N_VEC];                /* input vectors */
   double x_partial[SZ_WORK], y_partial[SZ_WORK];  /* a chunk of each vector */
   double partial;
```

```
   int i, self, mtype, offset;
   MPI_Status status;

   MPI_Init(&argc, &argv);              /* initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &self);

   if (self == MASTER) {                /* master process */

      generate(x, y);                   /* generates two vectors */

      mtype = FROM_MASTER;
      offset = 0;
         for (i = 1; i < N_PROC; i++) {          /* send messages to workers */
            MPI_Send(&x[offset], SZ_WORK, MPI_DOUBLE, i, mtype,
                  MPI_COMM_WORLD);
            MPI_Send(&y[offset], SZ_WORK, MPI_DOUBLE, i, mtype,
                  MPI_COMM_WORLD);
            offset += SZ_WORK;
         }
         mtype = TO_MASTER;
         for (i = 1; i < N_PROC; i++) {          /* receive messages from workers */
            MPI_Recv(&partial, 1, MPI_DOUBLE, i, mtype,
                  MPI_COMM_WORLD, &status);
            sum += partial;
         }
         printf("Inner product is %f\n", sum);
   }
   else {                                  /* worker process */
         mtype = FROM_MASTER;              /* receive a message from master */
         MPI_Recv(x_partial, SZ_WORK, MPI_DOUBLE, MASTER, mtype,
                        MPI_COMM_WORLD, &status);
         MPI_Recv(y_partial, SZ_WORK, MPI_DOUBLE, MASTER, mtype,
                        MPI_COMM_WORLD, &status);
         partial = worker(SZ_WORK, x_partial, y_partial);
         mtype = TO_MASTER;
         MPI_Send(&partial, 1, MPI_DOUBLE, MASTER, mtype,
                                 MPI_COMM_WORLD);
   }                                       /* send result back to master */

   MPI_Finalize();
}
```

Distributed Shared Memory Systems.

As mentioned previously, distributed shared memory systems attempt to unify the message passing and shared memory programming models. Since DSM's span both

physically shared and physically distributed memory systems, DSM's are also concerned with the interconnection network that provide the data to the requesting processor in an efficient and timely fashion. Both the bandwidth (amount of data that can be supplied in a unit time) and latency (the time it takes to receive the first piece of requested data from the time the request is issued) are important to the design of DSM's. Precisely because of the generally longer latencies encountered in large scale DSM's, multithreading has received considerable attention; multithreading model can be utilized to tolerate (or mask) memory latencies. In this paper we will not address issues related to interconnection networks or latency tolerance techniques.

Given that the overall objective of DSM's is to provide cost-effective mechanisms to manage the extended memory space across multiple levels of hierarchy, the design space is huge. In this tutorial will address some of the more important classes in the design space. In this section we will overview the major issues that face the design of DSM's. To start, the management of large logical memory space involves moving data dynamically across the memory layers of a distributed system. This includes the mapping of the user data to the various memory modules. The data may be uniquely mapped to a physical address (as done in cache coherent systems) or replicating the data to several physical addresses (as done in reflective memory systems and, to some extent, in cache-only systems). Even in uniquely mapped systems, data may be replicated in lower levels of memories (i.e., caches). Replication, in turn requires means for maintaining consistency. Directories are often the key to tracking the multiple copies and play a key role in coherency of replicated data. . The coherency can be maintained by hardware or software.

The granularity of the data that is shared and moved across memory hierarchies is another design consideration. The granularity can be based on objects without semantic meaning, based purely on a sequence of bytes (e.g., a memory word, a cache block, a page) or it can be based on objects with semantic basis (e.g., variables, data structures or objects in the sense of object-oriented programming model). Hardware solutions often use finer grained objects (often without semantic meaning) while software implementations rely on coarser grained objects.

The design trade-offs in managing shared memory, be it in hardware or software, depend on the freedom (or limitations) provided by the semantics of the programming model. This leads to issues such as the order in which memory accesses performed by individual processors to individual memory modules and their interactions with memory accesses performed by other processors. The most restrictive programming semantics (known as *Sequential Consistency*) requires that (1) the memory accesses of each individual processor be executed in the order defined by the sequence defined by the program executed on the processor and (2) memory accesses of different processors be executed in some interleaved fashion. In order to utilize modern processors as the building blocks in a DSM, sequential consistency model is often not guaranteed by the underlying system, placing the onus on the programmer in maintaining the necessary order of memory accesses. There has been a considerable research into the various memory consistency

semantics and the trade-offs they present in terms of performance and the programming complexity.

Synchronization and coordination among concurrent computations (i.e., processes or threads) in shared memory programming relies on the use of mutual exclusion, critical sections and barriers. Implementation of the mechanisms needed for mutual exclusion (often based on the use of locks) and barriers explore the design space spanning hardware instructions (such as test&set, Load-Linked and Store-Conditional instructions, fetch&operate, combining networks), spin locks (including the use of shadow locks), Queue or Array locks, sense-reversing barriers and tree-barriers. These solutions explore trade-offs between the performance (in terms of latency, serialization, network traffic) and scalability.

Related to the above set of issues, resource management, particularly in terms of minimizing data migration, thread migration, messages exchanged are all significant in achieving commercial cost-performance ratios in making DSM's as contenders. In section II, we will detail issues related to data-coherency, memory consistency models, and the implementation of locks and barriers. In section III, we will case-study 3 hardware DSM implementations, representing cache-coherent architectures, cache-only architectures and reflective memories. In section IV, we will describe software DSM implementations covering page-based, variable-based and object-based data models. In section V, we will describe how DSM's can be implemented using both hardware and software techniques. In section VI, we conclude the survey with a set of open isuse, and a prognosis on the DSM's commercial viability, and alternatives to DSMs.

2. Issues in Shared Memory Systems

In this section we will concentrate on 3 main issues in the design of hardware or software based distributed shared memory system. They are related to data coherence, memory consistency and synchronization.

2.1.Data Coherency.

The use of cache memories is so pervasive in today's computer systems it is difficult to imagine processors without them. Cache memories, along with virtual memories and processor registers form a continuum of memory hierarchies that rely on the principle of locality of reference. Most applications exhibit temporal and spatial localities among instructions and data. Spatial locality implies that memory locations that are spatially (address-wise) near the currently referenced address will likely be referenced. Temporal locality implies that the currently referenced address will likely be referenced in the near future (time-wise). Memory hierarchies are designed to keep most likely referenced items in the fastest devices. This results in an effective reduction in access time.

2.1.1.Cache Coherency.

The inconsistency that exists between main memory and write-back caches does not cause any problems in uniprocessor systems. But techniques are needed to ensure that consistent data is available to all processors in a multiprocessor system. Cache coherency can be maintained either by hardware techniques or software techniques. We will first introduce hardware solutions.

2.1.2. Snoopy Protocols .

These protocols are applicable for small scale multiprocessors systems where the processors are connected to memory via a common bus, making the shared memory equally accessible to all processors (also known as Symmetric Multiprocessor systems SMP, Uniform Memory Access systems, UMA). In addition to the shared memory, each processor contains a local cache memory (or multi-level caches). Since all processors and their cache memories (or the controller hardware) are connected to a common bus, the cache memories can *snoop* on the bus for maintaining coherent data. Each cache line is associated with a state, and the cache controller will modify the states to track changes to cache lines made either locally or remotely. A hit on a read implies that the cache data is consistent with that in main memory and copies that may exist in other processors' caches. A read miss leads to a request for the data. This request can be satisfied by either the main memory (if no other cache has a copy of the data), or by another cache which has a (possibly newer) copy of the data. Initially, when only one cache has a copy, the cache line is set to *Exclusive* state. However, when other caches request for a read copy, the state of the cache line (in all processors) is set to *Shared*.

Consider what happens when a processor attempts to write to a (local) cache line. On a hit, if the state of the local cache line is *Exclusive* (or *Modified*), the write can proceed without any delay, and state is changed to *Modified*. This is because, Exclusive or Modified state with the data guarantees that no copies of the data exist in other caches. If the local state is *Shared* (which implies the existence of copies of the data item in other processors) then an invalidation signal must be broadcast on the common bus, so that all other caches will set their cache lines to *Invalid* state. Following the invalidation, the write can be completed in local cache, changing the state to *Modified*.

On a write-miss request is placed on the common bus. If no other cache contains a copy, the data comes from memory, the write can be completed by the processor and the cache line is set to *Modified*. If other caches have the requested data in *Shared* state, the copies are invalidated, the write can complete with a single *Modified* copy. If a different processor has a *Modified* copy, the data is written back to main memory and the processor invalidates its copy. The write can now be completed, leading to a *Modified* line at the requesting processor. Such snoopy protocols are sometimes called MESI, standing for the names of states associated with cache lines: Modified, Exclusive, Shared or Invalid. Many variations of the MESI protocol have been reported [Baer]. In general the performance of a cache coherency protocol depends on the amount of sharing (i.e., number of shared cache blocks), number of copies, number of writers and granularity of sharing.

Instead of invalidating shared copies on a write, it may be possible to provide updated copies. It may be possible with appropriate hardware to detect when a cache line is no longer shared by other processors, eliminating update messages. The major trade-off between update and invalidate technique lies in the amount of bus traffic resulting from the update messages that include data as compared to the cache misses subsequent to invalidation messages. Update protocols are better suited for applications with a single writer and multiple readers, while invalidation protocols are favored when multiple writers exist.

2.1.3. Directory Protocols

Snoopy protocols rely on the ability to listen to and broadcast invalidations on a common bus. However, the common bus places a limit on the number of processing nodes in a SMP system. Large scale multiprocessor and distributed systems must use more complex interconnection mechanisms, such as multiple buses, N-dimensional grids, Crossbar switches and multistage interconnection networks. New techniques are needed to assure that invalidation messages are received (and acknowledged) by all caches with copies of the shared data. This is normally achieved by keeping a directory with main memory units. There exists one directory entry corresponding to each cache block, and the entry keeps track of shared copies, or the identification of the processor that contains modified data. On a read miss, a processor requests the memory unit for data. The request may go to a remote memory unit depending on the address. If the data is not modified, a copy is sent to the requesting cache, and the directory entry is modified to reflect the existence of a shared copy. If a modified copy exists at another cache, the new data is written back to the memory, a copy of the data is provided to the requesting cache, and the directory is marked to reflect the existence of two shared copies.

In order to handle writes, it is necessary to maintain state information with each cache block at local caches, somewhat similar to the Snoopy protocols. On a write hit, the write can proceed immediately if the state of the cache line is Modified. Otherwise (the state is Shared), Invalidation message is communicated to the memory unit, which in turn sends invalidation signals to all caches with shared copies (and receive acknowledgements). Only after the completion of this process can the processor proceed with a write. The directory is marked to reflect the existence of a modified copy. A write miss is handled as a combination of read-miss and write-hit.

Notice that in the approach outlined here, the directory associated with each memory unit is responsible for tracking the shared copies and for sending invalidation signals. This is sometimes known as p+1 directory to reflect the fact that each directory entry may need p+1 bits to track the existence of up to p read copies and one write copy. The memory requirements imposed by such directory methods can be alleviated by allowing fewer copies (less than p), and the copies are tracked using "pointers" to the processors containing copies. Whether using p+1 bits or fixed number of pointers, copies of data items is maintained by (centralized) directories associated with memory modules. We can consider distributing each directory entry as follows. On the first request for data, the memory unit (or the home memory unit) supplies the requested data, and marks the

directory with a "pointer" to the requesting processor. Future read requests will be forwarded to the processor which has the copy, and the requesting processors are linked together. In other words, the processors with copies of the data are thus linked, and track all shared copies. On a write request, an invalidation signal is sent along the linked list to all shared copies. The home memory unit can wait until invalidations are acknowledged before permitting the writer to proceed. The home memory unit can also send the identification of the writer so that acknowledgements to invalidations can be sent directly to the writer. Scalable Coherence Interface (SCI) standard uses a doubly linked list of shared copies. This permits a processor to remove itself from the linked list when it no longer contains a copy of the shared cache line.

Numerous variations have been proposed and implemented to improve the performance of the directory based protocols. Hybrid techniques that combine Snoopy protocols with Directory based protocols have also been investigated in Stanford DASH system. Such systems can be viewed as networks of clusters, where each cluster relies on bus snooping and use directories across clusters (see Cluster Computing).

The performance of directory based techniques depend on the number of shared blocks, number of copies of individual shared blocks, if multicasting is available, number of writers. The amount of memory needed for directories depend on the granularity of sharing, the number of processors (in p+1 directory), number of shared copies (in pointer based methods).

2.1.4. Software Based Coherency Techniques.

Using large cache blocks can reduce certain types of overheads in maintaining coherence as well as reduce the overall cache miss rates. However, larger cache blocks will increase the possibility of false-sharing. False sharing refers to the situation when 2 or more processors which do not really share any specific memory address, however they appear to share a cache line, since the variables (or addresses) accessed by the different processors fall to the same cache line. Compile time analysis can detect and eliminate unnecessary invalidations in some false sharing cases.

Software can also help in improving the performance of hardware based coherency techniques described above. It is possible to detect when a processor no longer accesses a cache line (or variable), and “self-invalidation” can be used to eliminate unnecessary invalidation signals. In the simplest method (known as Indiscriminate Invalidation), consider an indexed variable X being modified inside a loop. If we do not know how the loop iterations will be allocated to processors, we may require each processor to read the variable X a the start of each iteration and flush the variable back to memory at the end of the loop iteration (that is invalidated). However, this is unnecessary since not all values of X are accessed in each iteration, and it is also possible that several, contiguous iterations may be assigned to the same processor.

In Selective invalidation technique, if static analysis reveals that a specific variable may be modified in a loop iteration, the variable will be marked with a "Change Bit". This

implies that at the end of the loop iteration, the variable may have been modified by some processor. The processor which actually modifies the variable resets the Change Bit since the processor already has an updated copy. All other processors will invalidate their copies of the variable.

A more complex technique involves the use of Version numbers with individual variables. Each variable is associated with a Current Version Number (CVN). If static analysis determines that a variable may be modified (by any processor), all processors will be required to increment the CVN associated with the variable. In addition, when a processor acquires a new copy of a variable, the variable will be associated with a Birth Version Number (BVN) which is set to the current CVN. When a processor actually modifies a variable, the processor will set the BVN to the CVN+1. If the BVN of a variable in a processor is greater than the CVN of that variable, the processor has the updated value; otherwise, the processor invalidates its copy of the variable.

Migration of processes or threads from one node to another can lead to poor cache performances since the migration can cause "false" sharing: the original node where the thread resided may falsely assume that cache lines are shared with the new node to where the thread migrated. Some software techniques to selectively invalidate cache lines when threads migrate have been proposed.

Software aided prefetching of cache lines is often used to reduce cache misses. In shared memory systems, prefetching may actually increase misses, unless it is possible to predict if a prefetched cache line will be invalidated before its use.

Summary:

2.2. Memory Consistency

While data coherency (or cache coherency) techniques aim at assuring that copies individual data items (or cache blocks) will be up to date (or copies will be invalid), a consistent memory implies that view of the entire shared memory presented to all processors will be identical. This requirement can also be stated in terms of the order in which operations performed on shared memory will be made visible to individual processors in a multiprocessor system. In order to understand the relationship between the ordering of memory operations and memory consistency, consider the following example with two processors P1 and P2. P1 performs a write to a shared variable X (operation-1) followed by a read of variable Y (operation-2); P2 performs a read of variable X (operation-3) followed by a write to shared variable Y (operation-4). For each processor, we can potentially consider 4! different orders for the four operations. However, we expect that the order in which each processor executes the operations (i.e., program order) be preserved. This requires that operation-1 always be executed before opeariont-2; operation-3 before operation-4. Now we have only 6 possible orders in which the operations can appear.

While it is possible for the two processors to see the operations in different order, intuition tells us that "correct" program behavior requires that all processors see the memory operations performed in the same order. These two requirements (Program Order be preserved, and all processors see the memory operations in the same order) are used to define a correct behavior of concurrent programs, and is termed as Sequential Consistency of memory.

Ordering-1	Ordering-2	Ordering-3	Ordering-4	Ordering-5	Ordering-6
P1: Write X	P1: Write X	P1: Write X	P2: Read X	P2: Read X	P2: Read X
P1: Read Y	P2: Read X	P2: Read X	P1: Write X	P2: Write Y	P1: Write X
P2: Read X	P1: Read Y	P2: Write Y	P1: Read Y	P1: Write X	P2: Write Y
P2: Write Y	P2: Write Y	P1: Read Y	P2: Write Y	P1: Read Y	P1: Read Y

2.2.1. Sequential Consistency.

Sequential consistency may also be described as follows. In addition to preserving program order of each concurrent program, the behavior should be the same as if the program is executed on a single processor by interleaving the operations of the concurrent program segments. The order in which all memory operations appear to all processors is then given by the interleaved execution.

Although appealing conceptually, Sequential consistency can result in very poor performance on modern processing systems. Some cause can be understood by examining modern architectural features that must be thwarted to guarantee sequential consistency. Consider processors with Cache memories. In such systems, it is not only sufficient to make sure a write is completed (written to local cache and even the main memory), but we need make sure that this write is "seen" by all processors and they either update or invalidate their caches. To guarantee this, we need to wait for an acknowledgement from all cache copies before assuming that a write was made visible to all processors. Such a requirement will significantly affect the performance of write operations. Likewise, the use of buffers for writes cannot be utilized if Sequential Consistency must be guaranteed.

In order to improve the performance of concurrent processing systems, we may not require that the hardware guarantee the second requirement of maintaining a common ordering among memory operations that is seen by all processors, but assure sequential consistency by other means (e.g., programming constructs). Consider one such relaxation on hardware requirements. In Process consistency, different processors of a parallel processing system may see different ordering among memory operations; however, the program order is still preserved. In our previous example, it is now possible for processor P1 to see a different ordering among the four memory operations (say Ordering 1) while P2 sees a different ordering (say Ordering 5). This is troubling because, now P1 and P2 will obtain different values for X and Y. In order to assure a correct program (as per Sequential Consistency), it is now necessary to use programming constructs such as locks (or synchronization) before allowing a processor to modify shared memory. In other words, it is now the responsibility of the programmer to define a sequential order among

memory accesses for shared variables. This is normally done by defining critical sections, and the use of mutual-exclusion to enter critical sections.

2.2.2. Weak Ordering.

Once we accept that it is programmer responsibility to specify "serialization" of accesses shared variables, we can further reduce the restrictions on what the hardware must preserve. That is the basis of all Weak consistency model.

1. Accesses to synchronization variables is **sequentially consistent**

2. No access to a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.

3. No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

The first condition forces a global order on all synchronization variables. Since ordinary variables are accessed only in critical sections (after accessing synchronization variables), Sequential consistency is assured even on ordinary variables.

The second condition implies that before a synchronization variable is released (and subsequently obtained by a different processor), all accesses made to ordinary variables are made globally visible to all processors. Likewise, the third condition requires that before a processor can access ordinary variables inside a critical section, accesses to synchronization variables must be globally visible. This forces mutual exclusion on synchronization variables and makes changes made in previous critical sections are globally visible.

2.2.3. Release Consistency.

While using synchronization variables (in Weak ordering), we normally associate locks with synchronization variables. We use lock and unlock (or acquire and release) operations on these locks. When you acquire, you have not yet made any changes to shared variables (other processes may have). When you release a lock, you may have updated variables and these variables must be made available to other processors. So, we need to maintain consistent data only on a lock release. How does the performance improve? You are not guaranteeing consistency memory until a lock is released

1. Before an ordinary variable is accessed, all previous acquires of synchronization variables performed by the processor must have completed.

2. Before a release on a synchronization variable is allowed, all previous reads and writes on ordinary variables performed by the processor must have completed.

3. The acquire and release accesses must be processor consistent.

Notice that since we assume mutual exclusion on acquires, we do not require Sequential consistency on acquires.

Program correctness is assured by the programmer by using acquires and releases in proper order and accessing ordinary variables in critical section.

The performance of Release consistency can be improved using "Lazy" release whereby, the shared memory is made consistent only on acquire by a different processor.

2.2.4. Entry and Scope Consistency

In both Weak ordering and Release consistency models, the shared memory is made consistent when any synchronization variable is released (or accessed). However, if we can associate a set of shared variable with each synchronization variable, then we only need to maintain consistency on these variables when the associated synchronization variable is released (or accessed). Entry consistency requires that the program specify the association between shared memory and synchronization variables. Scope consistency is similar to Entry consistency, but the associations between shared variables and synchronization variables is implicitly extracted. Consider the following example where lock-1 and lock-2 are synchronization variables while A,B,C, and D are ordinary shared variables.

```
P1                                  P2

Lock lock-1
        A =1
        Lock lock-2
                B =1
        Unlock lock-2
Unlock l2
                                    Lock lock-2
                                    C = A    ----- may not see A=1
                                    D = B
                                    Unlock lock-2
```

A similar effect will be achieved in Entry consistency by associating lock-1 with variable A and lock-2 with variables A, B, C and D.

2.2.5. Hardware Prefetch and Speculative Loads.

Relaxed memory consistency models discussed here increase the programming complexity, either by requiring the programmer to carefully and correctly use synchronization among the processors, or by requiring the compiler to insert necessary barrier instructions (e.g., Memory Barrier and Write Barrier) to assure program correctness. A recent study indicated that while it is difficult to outperform relaxed

memory models, two mechanisms can substantial improve the performance of Sequential consistency hardware [Pai and Adve's paper].

a). Hardware prefetch for write. Here, a request for Exclusive access of a cache block is issued even before the processor is ready to do a write, overlapping the time for invalidations and acknowledgements needed to implement sequential consistency correctly, with other computations. However, improper use of the prefetch may unnecessarily invalidate cache copies at other processors and this may lead to increased cache misses.

b). Speculative loads. Here cache blocks for load are prefetched, without changing any exclusive accesses that are currently held by other processors. If the data is modified before it is actually used, the prefetched copies are invalidated. Otherwise, one can realize performance gains from the prefetch.

Summary. It appears that weak ordering is the dominating memory consistency that is supported by current processors that are using in a multiprocessor systems. Sequential consistency is assured implicitly and transparently by software layers or by the programmer. In addition to improved performance, weak ordering may also have benefits in supporting fault-tolerance to applications using DSM systems, in terms of the amount state information that must be checkpointed [Hecht 99].

2.3. Support For Synchronization.

In the previous section we assumed that the programmer relies on synchronization among the processors (or processes) while accessing shared memory in order to assure program correctness as defined by the Sequential consistency model. The two fundamental synchronization constructs used by programmers are mutual exclusion locks and barriers. Mutual exclusion locks can be acquired by only one processor at a time, forcing a global sequential order on the locks. When barriers are used a processor is forced to wait for its partners and proceed beyond the barrier only when all partners reach the barrier. In this section we describe how mutual exclusion locks and barriers can be supported in hardware or software, and discuss performance of various implementations.

2.3.1 Mutual Exclusion Locks.

In the simplest implementation, mutual exclusion can be achieved using an atomic instruction that sets a flag if it is currently reset, otherwise, fails. Such Test-and-Set instructions were implemented in many older machines that supported multi-tasking (or time-shared) operating systems. With the advent of cache memories (and associated re-ordering of memory accesses), it become extremely inefficient to implement atomic Test-and-Set instructions. Modern processors actually use two separate instructions to achieve the atomicity. Consider the following example that uses Load Linked (LL) and Store Conditional (SC) instructions. The SC to the same memory location as the LL will be successful only if there are no intervening memory accesses to the same memory location. Otherwise, the SC fails to modify the memory location.

```
Try:    Move R3, R4         ; Move value to be exchanged
        LL R2, 0(R1)        ; load linked to memory
        SC R3, 0(R1)        ; Store conditional to the same memory location
        BEQZ R3, Try        ; if unsuccessful, try again
        Move R4, R3
```

In this example, the value in R3 will be stored only if SC is successful and the value in R3 will be non-zero. Otherwise, SC will fail to change the value in memory and R3 will be set to zero.

Typically, LL stores the memory address in a special Link Register which is compared with the memory address of a subsequent SC instruction. The Link Register is rest by any memory accesses to the same address by other processors or on a context switch of the current process. In a multiprocessor system, LL and SC can be implemented using cache coherency techniques previously discussed. For example, in snoopy systems, the Link Register is reset by snooping on the shared bus.

Notice that in this example we are actually using "Spin Locks" where a processor is not blocked on an unsuccessful attempt to acquire a lock. Instead, the unsuccessful processor will repeat its attempt to acquire the lock.

2.3.1.1. Shadow locks.

So far we have been assuming that the coherency of mutual exclusion lock variable is guaranteed, which can significantly reduce the performance of shared memory systems. Consider how the repeated attempts to acquire a spin-lock can lead to repeated invalidations of the lock variable. In order to improve the performance, processors are required to spin on a "shadow" lock. All spinning processors try to acquire the lock by accessing the lock variable in common memory. Unsuccessful processor will cache the "locked-value" in local caches and spin on local copies. The local copies are invalidated when the lock becomes available.

Consider the following code segment for an implementation of the spin locks

```
Lockit:     LL R2, 0(R1)            ; Load Linked
            BNEZ R2, Lockit         ; not available, spin
            Load Immediate  R2, #1  ; locked value
            SC R2, 0(R1)            ; store
            BEQZ R2, Lockit         ; branch if unsuccessful
```

The first branch (BNEZ) does the spin. The second branch is needed for atomicity.

2.3.1.2. Other Variations.

Spinning on a single variable causes excessive network traffic since all unsuccessful processors attempt acquire the lock as soon as it becomes available. We can consider implementation of "exponential-back-off" techniques whereby a processor uses different amounts of delays while attempting to acquire a lock. Alternatively, we can associate an "Array" for lock variables so that each processor spins on a different array element. A similar effect can be achieved using "Ticket" locks. Here, an integer (or ticket number) is assigned to each unsuccessful processor. The value of the ticket being serviced is incremented on each release of the lock, and a processor with a matching ticket number will then be allowed to acquire the lock. This technique is similar to how customers are assigned a service number and they are serviced when the current number serviced matches their number.

In Queue locks and Granting Locks, the unsuccessful processors are linked together so that the current lock holder can release the lock to the next processor in the queue. Notice that these techniques are similar to the techniques used in older systems where an unsuccessful process is blocked and queued on the lock by the operating system.

2.3.2. Barrier Synchronization.

In addition to mutual exclusion, shared memory parallel programming requires coordination among the current processes (or threads), whereby processes are required to rendezvous periodically before proceeding with computation. Such coordination is traditionally achieved using Fork and Join constructs. Before forking parallel computations, an integer value indicating the number of concurrent processes is set in aj join-variable. As processes complete their assigned work, they execute the Join operation which atomically decrements the current value of the join-variable. A value of zero indicates that all concurrent processes have completed their tasks. In order to implement such barriers (i.e., Join construct), we need to use two mutual exclusion locks - one to decrement the counter and one to force processes to wait at the barrier

How would we implement the join (barrier) using the atomic instructions we have seen so far. We need two locks: first acquire the join variable -- and increment; and one to make sure all processes wait until the last process arrives at the barrier. Consider the following implementation.

```
.Lock (counter_lock);                  /*to make sure updates are atomic
If (counter == p) release = 0;         /* first reset release
                        /* this is to make sure that processes  are forced to wait
count = count-1;
unlock(counter_lock);
If (count ==0) {
        count = p;                     /* after all arrive, reset the counter
        release = 1; }                 /* and release waiting processes
else    spin (release = 1);            /* otherwise spin until release
```

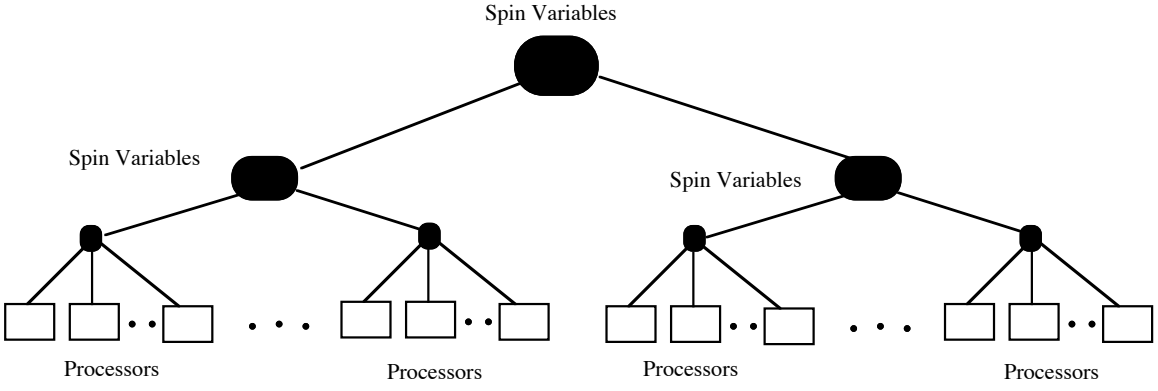
The above implementation is not adequate in applications containing several rendezvous points. Consider the following situation. There are p processes joining at a barrier. Normally, these processes repeat the join again at a different point in computation. The process which arrives last at the barrier is context-switched out, while one of the released process completed its next assigned task and returns to the next barrier, resetting the release flag. When the context-switched process is restarted, it is blocked since the release flag is reset, leading to a deadlock since only p-1 processes join at the next barrier. A suggested remedy to this situation, known as "sense reversing barrier" uses alternating values for the reset flag: 0 one time and 1 the next time to indicate a release status.

```
local_sense = ! local_sense;          /* reverse local sense

.Lock (counter_lock);                       /*to make sure updates are atomic
If (counter ==p)                      /* first reset release
        release = ! local_sense;            /* this is to make sure that processes
                                      /* are forced to wait
count = count-1;
unlock(counter_lock);
If (count ==0) {
        count = p;                    /* after all arrive, reset the counter
        release = local_sense; }      /* and release waiting processes
else    spin (release = local_sense); /* otherwise spin until release
```

Tree-structured and other variations to barriers.

In the implementations presented above, all processes are spinning on the same "release" variable causing a significant amount of network traffic. Even if shadow variables are used, the invalidations on a release can degrade the performance. Several researchers have proposed more complex approaches for the implementation of barriers to reduce the number of messages. Consider a tree-structured algorithm where processors (or processes) are required to spin on different variables.



They are divided into groups and each group spins on a separate variable. The last processor in each group to arrive at the barrier must "report" to a higher level in the tree and spin on a new variabl. When we reach the root of the tree which indicates the

completion of the barrier, the children of the root are released. These children in turn release their children, propagating the release down the tree.

One problem with this approach is that it is not possible to statically know on which variable to spin, since it is not known which processor moves up the tree and spins on a new variable. This may cause performance penalties since localities can be maintained on the spin variables. Variations of the tree barriers (e.g., Butterfly, Dissemination and Tournament barriers) statically determine which processor moves up the tree.

A second problem with tree barriers is related to the serialization delays where the processor that is responsible for propagating the release signal down the tree will be forced to wait until all its children are released. In a variation to the tree-barriers [Mellor-Crummy and Scott], it is possible to consider using two separate trees, one for spinning and one for releasing processors. It is then possible to designate different processors as the roots in the two different trees and to use different fan-in and fan-out for the different trees. Smaller fan-out for the release tree avoid long serialization delays.

Summary.