Spin-Lock Synchronization on the Butterfly and KSR1

Xiaodong Zhang and Robert Castañeda
University of Texas at San Antonio
Elisa W. Chan
Emis Software

The execution behavior of spin-lock algorithms is significantly different between architectures based on multistage interconnection networks and those based on hierarchical rings. These tests suggest how spin-locks can be made cost-effective on both.

Parallel processing on shared-memory multiprocessors often requires that a single processor have exclusive access to shared data structures (critical variables) or serial portions of a program (critical sections). All such systems provide hardware primitives for exclusive access to critical variables, usually in the form of instructions that atomically read and then write to a single memory location. Many systems also provide a software protocol that takes more than one instruction and specifies how to acquire and release a lock that guards the critical section, guaranteeing mutual exclusion for the code.

The protocol sets a lock before a processor accesses a critical variable or section, and other processors trying to enter the variable or section must wait until the lock is released. A simple spin-lock is most common, but this method wastes processor cycles during delays. Moreover, a spin-wait can generate extra traffic and consume communication bandwidth on the network. This slows processors doing useful work, including the processor working in the critical section.

The drawbacks of the simple spin-lock limit its effective use to small critical sections. Applications with large critical sections and a large number of processors require more efficient algorithms to minimize processor and network overheads. Variations on the spin-lock[1] have been tested on the Sequent Symmetry, a bus-based shared-memory multiprocessor. Algorithms for scalable synchronization[2] have also been tested on the BBN Butterfly I, a large-scale shared-memory multiprocessor with a multistage interconnection network (MIN).

1063-6552/94/ $4.00 © 1994 IEEE

We have extended the investigation to the BBN GP1000 and TC2000, both MIN-based multiprocessors with network contention heavier than that on the Butterfly I. We have also implemented algorithms on Kendall Square Research's KSR1, a hierarchical-ring (HR) multiprocessor system, to study the effects of cache coherence. (Sidebars discuss the architecture of the Butterfly systems and the KSR1.) The execution behavior of spin-lock algorithms is significantly different between MIN-based and HR-based architectures. Our tests suggest that HR-based architectures handle network and memory contention more efficiently than MIN-based architectures. However, our results also suggest how spin-locks can be made cost-effective on both.

Centralized spin-lock algorithms

According to how they allocate locks, we classify spin-locks into *centralized* and *distributed* algorithms. A centralized algorithm defines a globally shared lock that is

Butterfly systems

The MIN-based Butterfly systems connect multiple processors to multiple memory modules through an interconnection network (Figure A shows a typical two-stage system). The BBN GP1000 is a MIMD system with up to 256 16-MHz Motorola 68020 processors, connected via a network of 4×4 switches.[1] It uses the same network as the BBN Butterfly I, the company's first generation of MIN-based multiprocessor systems. The major difference in architecture is the processor; the Butterfly I uses 8-MHz Motorola 68000 processors. In both systems, the bandwidth of each switch path is 4 Mbytes per second.

According to experiments with an analytical model, if the network speed remains the same and the processor speed is doubled, the potential network contention in a computation is at least doubled.[2] In other words, the overhead caused by network contention on the GP1000 is at least two times greater than for the same computation on the Butterfly I.

The TC2000, the latest and most powerful BBN system, supports up to 512 20-MHz Motorola 88100 processors.[3] The network uses a butterfly switch composed of 8×8 switches. The bandwidth of each switch path is 4.75 Mbytes per second, but network speed is still not sufficient to reduce network contention. Each node has 16 Mbytes of memory that can be accessed by any processor in the system, and a Motorola 88200 paged memory management unit for virtual memory processing. Also, each node has a 32-Kbyte code cache and a 16-Kbyte data cache. The TC2000 avoids the cache-coherence problem by not allowing caching of shared data. It caches private data, read-only shared data, and instructions, while references to modifiable shared data bypass the cache. To reduce memory contention in applications, the TC2000 offers an option of an interleaved shared-memory scheme.

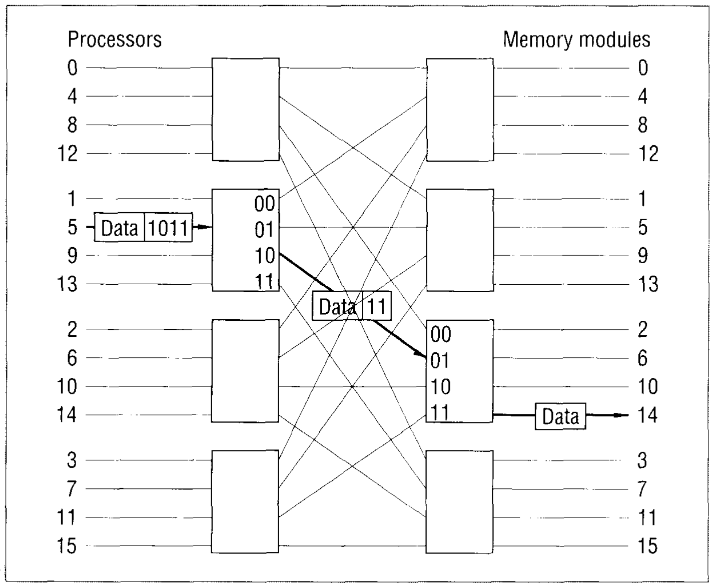


Figure A. A two-stage MIN-based network.

References

1. BBN Advanced Computer, *Inside the GP1000*, 1989.
2. X. Zhang and X. Qin, "Performance Prediction and Evaluation of Parallel Processing on a NUMA Multiprocessor," *IEEE Trans. Software Eng.*, Vol. 17, No. 10, Oct. 1991, pp. 1059–1068.
3. BBN Advanced Computer, *Inside the TC2000*, 1989.

```
/* to initialize the lock */
Int Lock = 0;

/* to set the lock */
While (Test_and_Set (Lock, 1)) Delay ();

/* to release the lock */
Lock = 0;
```

Figure 1. The simple spin-lock and its variations.

```
/* to initialize the two locks */
Int Ticket = 0, Current = 0;

/* to set the lock */
MyTicket = AtomicAdd(Ticket, 1);
While (MyTicket != Current) Delay (MyTicket – Current);

/* to release the lock */
Current = Current + 1;
```

Figure 2. The ticket lock.

allocated in a single processor to control multiprocessor access to the critical section. The hardware primitive usually provided by shared-memory multiprocessor systems consists of instructions that atomically read and then write to a single memory location. Using these instructions, we can construct a wide variety of centralized spin-lock algorithms.

Simple spin-lock and its variations

When a processor gets a simple spin-lock (see Figure 1), it immediately sets the lock busy and starts the atomic operations. It sets an unlock immediately after finishing work with the critical section. Other processors must busy-wait for the lock, using remote memory accesses through the network.

We can vary the simple spin-lock by applying different time delays between processors' attempts to access the lock variable:

- Simple spin-lock: Delay( ) = 0.
- Static-delay spin-lock: Delay( ) = *C*, a constant value.
- Exponential-delay spin-lock: Delay( ) increases exponentially with every failed attempt to acquire the lock.
- Computed-delay spin-lock: Delay( ) is some function of the estimated total spins versus the number of spins for the local processor.

The ticket lock

The ticket-lock algorithm derives dynamic delay information from the number of processors contending for a lock (see Figure 2).[2,3] The algorithm uses two lock variables:

- a ticket counter that is automatically incremented by a processor to obtain its ticket, and
- a current counter that indicates the number of the ticket that has access to the critical section.

After obtaining a ticket, each processor spins on the current lock until its turn arrives. Upon exiting the critical section, the processor increments the current lock to let the next ticket holder into the critical section.

The algorithm's most attractive feature is that it can predict the delay function to reduce network traffic better than with fixed functions applied to the simple spin-lock. The algorithm can compute a processor's delay from the difference between its ticket and the current lock value: The difference is the number of processors that will enter the critical section before the processor that holds the ticket. Also, unlike the simple spin-lock and its variations, the ticket-lock algorithm guarantees FIFO service.

We classify the ticket-lock algorithm as centralized because the ticket-lock variable is globally shared and allocated in a single processor. This means that the algorithm still has the problem of global spin in all processors in the network.

Centralized algorithms on Butterfly systems

In the Butterfly systems, the simple spin-lock and its variations involve a global spin implemented mainly through a Test_and_Set hardware operation (plus a While software instruction):

```
While (Test_and_Set (Lock, 1))
```

Test_and_Set must take place in one cycle to prevent another processor from accessing and modifying the memory location before the current processor completes the operation. The requesting processor usually accomplishes this indivisibility by holding the network path until the cycle is completed. BBN Butterfly machines implement atomic operations such as Test_and_Set in a remote "fetch and operate" form: An arithmetic logic unit attached to each memory module remotely performs the atomic instruction. Each waiting processor must spin across the network, but this implementation is still more efficient than a traditional remote read and write, and it reduces network traffic.

In contrast, the global spin implemented in the ticket-lock algorithm is a pure software operation:

```
While (Myticket != Current)
```

Although it performs only read operations (and thus avoids the overhead of unnecessary invalidations in a cache-coherent machine, this While statement causes substantial memory and network contention. A noncache machine (GP1000) or a machine that caches only local data (TC2000) performs the testing operation by a remote read with two operations: The source node first sends the destination node a request to read the Current

The KSR1

Kendall Square Research's HR-based KSR1 is a shared-memory system with up to 1,088 64-bit custom superscalar RISC processors.[1] Each 20-MHz processor has a 32-Mbyte cache and a 0.5-Mbyte subcache, further subdivided into 0.25 Mbyte for instructions and 0.25 Mbyte for data. A Mach-based operating system supports parallel computing on the KSR1. A task begins with a single sequential flow of control — called a *p-thread* — and creates others to perform work in parallel. The KSR C runtime library manages p-threads and implements synchronization algorithms.[2]

The KSR1's basic structure is the slotted ring, which connects 32 processors with local caches (see Figure B). The system uses a two-level hierarchy to interconnect 34 rings. Ring bandwidth is divided into continuously circulating slots whose number equals the number of processors plus the number of routers connected to the upper ring. Hence, a standard KSR1 ring has 34 message slots: 32 for the processors and two for the directory cell connected to the level 1 ring. Each slot can be loaded with a packet or *subpage* comprising a 16-byte header and 128 bytes of data. A processor that is ready to transmit a message waits for an empty slot to rotate through its ring interface. A single bit in the slot header identifies an empty slot.

The KSR1 provides three levels of cache access and three different access times from a processor. The local cache directory provides local-cache access reference. Each local cache directory is connected to a cell interconnect (CI), which provides the interface between the processor and the ring. Level 0 directories provide internal-ring cache-access reference through the cache link and the internal-ring bus. Level 1 directories provide external-ring cache-access reference through the cache link, the internal-ring bus, and the upper level ring bus. A CI is also part of the AllCache processor-routing directory (APRD) cell. In addition, AllCache router and directory (ARD) cells collect and build the cache directories at level 0 and level 1. These cells route access requests dynamically to the point of reference. A ring's search engine comprises all APRD and ARD cells in the ring. The search engine searches directories, moves data, and maintains cache coherence. Thus, search operations are done by hardware at relatively high speed.

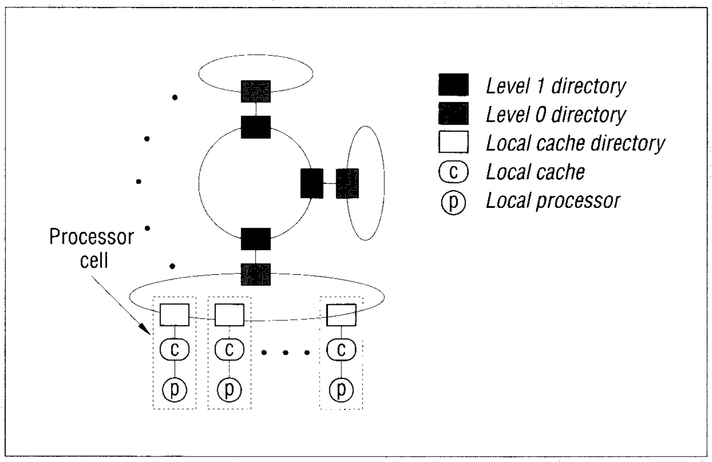


Figure B. An HR-based network.

A nonlocal data access starts when a local cache cannot satisfy a request from the local processor. The CI inserts a request into the ring when an empty slot is available. As the request passes each processor on the ring, each CI checks to see if the requested data is in its local cache. If so, the CI extracts the request and inserts a response, which continues around the ring to the requesting processor.

The ARD cell is the gateway to the next higher level ring. The cell contains a portion of the ring and the directory of the entire ring's data allocation. When a request reaches the cell, if the cell has an entry for the requested data, it forwards the message to the next processor. Otherwise, it routes the message to the next higher level ring.

A cache access in an internal ring requires a message to transit the entire ring. Similarly, a cache access to an external ring requires a message to transit three entire rings (the local ring, the upper level ring, and the external ring). Although the distance for accesses at the same level is the same number of circles, the number of searches required for each access may differ. This is because the number of searches depends on the distance between the requesting processor and the source processor.

The combination of all the local caches, the slotted rings, and the search engines forms the KSR1's AllCache memory system. The distributed ring architecture provides a single shared address space for all processors, making the KSR1 a cache-coherent system. Data can be transferred into a requesting processor's local cache for read and write on demand.

References

1. Kendall Square Research, *KSR1 Technology Background*, 1992.
2. Kendall Square Research, *KSR C Programming*, 1992.

value, and the destination node then sends the data back to the source node. This requires two message-passing operations. Because the destination node sends memory data back to the requesting node, another message is constructed and another path may be established.

Traffic flow in the switch on the GP1000 is duplex, but not bidirectional along the original path. The TC2000 improves remote-read performance by providing a single communication channel for the request and feedback data. Compared with network transactions for a remote read, the testing operation "!=" in a local processor is trivial. One remote test for global spin spends most of its time on the network. When many processors simultaneously check whether their MyTicket is equal to Current, all the requests try to establish one or more reading channels to the same memory location. This generates network contention at different levels of switches, and the two-channel communication on the GP1000 makes this contention even worse.

To compare network contention caused by simple spin-locks and ticket locks, we measured the average ratio between the real computing time attributed to Test_and_Set or the remote comparison in software, and the total computing time used in either operation, including network contention overhead.

Figure 3 gives the ratio curves for both operations on a GP1000 with 120 processors. With only one processor, the ratios for both are 1 because no network transactions are involved. As the number of processors increases, the ratios decrease for both operations because there is more network contention. However, the ratio for the software global spin decreases faster than for the hardware Test_and_Set, because it introduces more network contention.

Figure 4 shows that network contention in general is less on a TC2000 with 118 processors. The faster switching network and single communication channel for a remote-read operation make the performance of the software remote fetch-and-compare slightly better than the performance of Test_and_Set.

Software global spin might not significantly degrade the performance of a synchronization lock on a MIN-based architecture with relatively light network contention. For example, testing with a read is slightly more efficient than Test_and_Set on the Butterfly I,[2] because it has less network contention than the GP1000 and TC2000. However, if network contention is high, software global spin can severely degrade synchronization-lock performance, as we show later.

For simplicity, the ticket-lock algorithm in Figure 2 does not specify the delay function. In practice, we must carefully calculate its value:

$$Delay\,() = (MyTicket - Current) \times unit$$

where *unit* is the unit of time between a processor's entrance to and exit from the critical section:

$$unit = T(Acquire\_Lock) + T(critical\ section) + T(Release\_Lock) + T(network\ contention)$$

We can measure all the times except the network contention time, which is linearly proportional to the number of processors used and depends on how busy the network is when the ticket lock is performed. The value of *unit* is architecture dependent, and determining an optimal value for *unit* is important for achieving high ticket-lock performance.

Using our measurements for *T(Acquire_Lock)*, *T(critical section)*, and *T(Release_Lock)*, and our prediction for *T(network contention)*, we plotted the experimental opti-

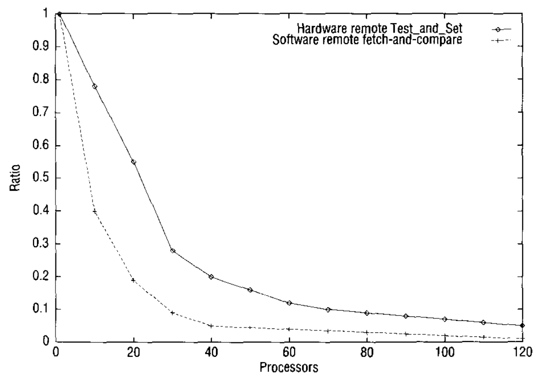


Figure 3. Ratios between the local and total computation of Test_and_Set and remote fetch-and-compare on the GP1000.

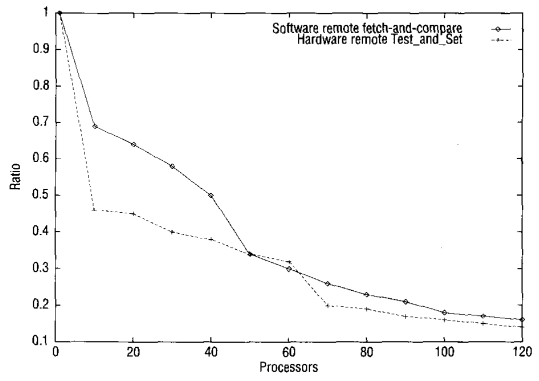


Figure 4. Ratios between the local and total computation of Test_and_Set and remote fetch-and-compare on the TC2000.

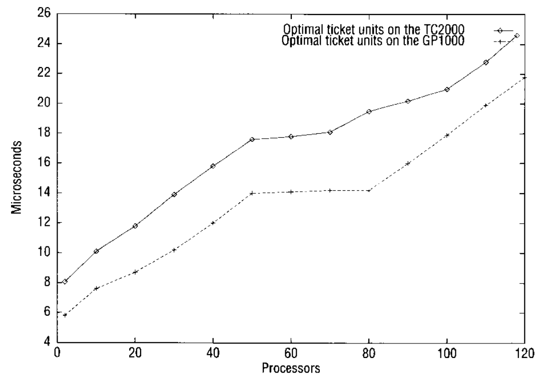


Figure 5. Optimal ticket *unit* versus number of processors on the GP1000 and the TC2000.

mal *unit* for the ticket lock on the GP1000 for up to 120 processors and on the TC2000 for up to 118 processors (see Figure 5). Without choosing an optimal unit in the ticket algorithm, experimental performance might not be as good as it should be.

Centralized algorithms on the KSR1

We can implement the simple spin-lock and its variations in two ways on the KSR1. If we want to use the instruction

```
While (Test_and_Set (Lock, 1))
```

we must simulate the Test_and_Set operation using related hardware primitives, because the KSR1 system does not directly support it. However, the KSR1 is a cache-coherent system, so write invalidations create a problem. The variable Lock is globally shared — each processor gets a copy in its local cache. Each access to Lock from each processor results in a read-hit (test) followed by a write-hit (set). After a write-hit, an invalidation message goes to all processors holding a copy of Lock. The invalidation transaction for each access of the lock per processor generates heavy network traffic and seriously degrades synchronization performance. The result is a fast "Ping-Pong" of invalidations whenever a processor accesses a lock.

A second spin-lock implementation defines an exclusively shared variable Lock in a subpage. The KSR1 provides two hardware instructions for locking a subpage:

- Gspwt (get subpage, wait): If a processor cannot immediately get access to the subpage it stalls until it can.
- Gspnwt (get subpage, no wait): A processor either gets access to the subpage or receives a message that it cannot have access at that time.

Processors conduct all accesses to Lock by trying to acquire the subpage. The current lock holder sets the subpage to an atomic state when it gets access. (Atomic state is a stronger form of ownership than the exclusive state, and the subpage enters and leaves it only as a result of explicit program requests.) Depending on their hardware instruction, all other requesting processors must either send requests again or wait (stall) until the subpage is released by the processor exiting the critical section. When the lock is released, the subpage goes to the next waiting processor on the ring, which sets it to an atomic state. Although the lock moves among the processors' caches, a large number of accesses to Lock from requesting processors can generate heavy network traffic.

Ticket lock

To take advantage of caches in a spin-lock, we can let spinning processors wait on the cache when the lock is busy. Only when the lock is free can they perform Test_and_Set and try to get the lock. Local-cache spinning does not consume network cycles while the lock is held. In the ticket lock on the KSR1, each processor spins while the lock is not free. The spin is implemented by

```
While (MyTicket != Current)
```

where MyTicket is a local variable and Current is globally shared.

Each processor performs a local read to check the ticket while the lock is busy, but the requirement of cache coherence results in extra overhead when the lock is released. Immediately after a processor releases a lock, the system must invalidate the cache lines in the cache copies of all processors waiting to read. Then each processor with a cache read-miss must fetch the new value and return to local spin. This is the first network transaction after the lock is released. Only one processor can get the lock and set it to busy. Any processor that has completed its read-miss before this happens will have seen the lock value as free. Hence, the second network transaction must invalidate the cache copies again. If the critical section is small, the overhead to ensure cache coherence is substantial. Again, a fast "Ping-Pong" reaction can occur whenever the lock is released by a processor.

Distributed spin-lock algorithms

To reduce contention in the network and in the memory module holding the lock, each processor should busy-wait only on a locally accessible variable for the lock. Similarly, the ticket lock in a cache-coherent system would be more efficient if the protocol invalidated or updated only the cache that gets the lock next.

A distributed algorithm can decentralize spin-locks throughout the memory modules in MIN-based and

HR-based systems.[4] In a distributed address space approach, each process views the local cache/memory as dedicated to itself, enabling the programmer to allocate data, schedule a process, and perform an operation in a specific physical processor. For example, to implement a distributed lock, a processor needs to know the address of the lock structure of its successor in the queue so it can directly update the lock of that processor after it releases the lock.

An array-based queuing lock does not use a single scalar variable to represent the lock; instead, it represents scalars as an array in which each element resides on a different node (see Figure 6).[1] As processors arrive at the lock, they atomically increment a counter to obtain the index of the array element on which to spin. Each processor spins locally until the processor that acquires the lock before it releases the lock and transfers control to it.

```
/* S is placed such that each element is on its own node */
Int S[] = {0, 1, 1, ..., 1}, Place = 0;

/* to set the lock and local spin */
MyPlace = AtomicAdd(Place, 1);
While(S[MyPlace % NumProcs()]);

/* to release the lock */
S[MyPlace % NumProcs()] = 1;
S[(MyPlace + 1) % NumProcs()] = 0;
```

Figure 6. Distributed algorithm for spin-lock in the array-based implementation.

Distributed algorithms on the Butterfly systems

The array-based distributed algorithm performs well on the Sequent Symmetry, a bus-based shared-memory multiprocessor.[1] However, it does not perform well on MIN-based multiprocessors that support shared memory through a distributed architecture (such as the Butterfly I).[2] Because the processors spin at statically unpredictable locations, the array-based queuing locks make it generally impossible to spin on "real" local locations. The performance of our implementation of the array-based algorithm on the GP1000 was also disappointing.

A new *list-based* queuing lock — called the MCS lock — has also been implemented on the Butterfly I.[2] In this scheme, every processor using the lock allocates a record containing a queue link and a Boolean lock flag. These links chain together processors that are holding or waiting for the lock, and each processor spins on its own locally accessible flag. Each processor in the queue holds the address of the record for the next processor so it can inform that processor when it releases the lock. Unfortunately, the MCS lock is implemented in MC68000 assembly language on the Butterfly I, so it is not portable to the BBN GP1000 or TC2000.

We implemented an *array-link-based* distributed lock (see Figures 7 and 8).[5] This is not a new distributed lock compared with the array-based queuing lock and the MCS list-based lock. However, our implementation is simpler and portable because it is written in C on the Mach 1000 operating system. It also combines both array and linked-list data structures, and explicitly distributes the lock to each processor.

The array-link-based distributed lock distributes an array data structure among the processors. The index value of the array in each processor is identical to the processor identification number. Each processor holds an element of the globally shared array. Another shared variable — Lock — provides FIFO service. As a processor arrives at the lock, it acquires the ID of the processor that acquired the lock right before it. This operation is done atomically. By keeping a record of the ID, the processor can enqueue itself and spin on a locally accessible flag in its own Lock_q structure. When a processor finishes work in the critical section, it sets the flag of the next processor in the queue and logically dequeues itself. (In fact, no real dequeue action is taken, which avoids extra network transactions.) This approach requires at most two remote-memory accesses waiting for the lock to be released.

Distributed algorithms on the KSR1

Because the KSR1 provides a sequentially consistent memory and programming model, it does not directly support distributed computing techniques. We therefore built a special data structure for implementing a distributed-lock algorithm. Each p-thread allocates a lock structure to a specific processor. The KSR1 sequentially generates the logical number of a p-thread starting from 0. Therefore, we can use an array to store the addresses of all the lock structures in the system. Array indices represent the p-thread logical numbers. A p-thread assigns the lock-structure address to the corresponding array entry after it allocates the lock structure in a processor. Through this globally shared array, each processor can access lock structures in other processors for linking itself in the queue and telling its successor to get the lock after release.

Performance on Butterfly systems

For fair comparison, our experiments on the GP1000 and TC2000 were similar to earlier experiments on the Butterfly I.[2] We measured the maximum time (the vertical axis) to acquire and release the lock with a given

```
Typedef Struct MLINK {
  Struct MLINK * Successor;
  Short Get_Lock;
} Mlink;
Short * Lock, NodeID;              /* Lock: local lock, NodeID: processor ID. */
Mlink * Lock_q[Num_Proc - 1];      /* Num_Proc: number of processors used */
Int NoDecount;

/* initialize the lock */
 *Lock = -1;
```

Figure 7. Basic data structures of the array-link-based distributed lock.

```
/* Acquire_Lock: processor NodeID acquires the Lock */
Acquire_Lock (NodeID, Lock)
  Short NodeID;
  Short *Lock;
  {Int Pred;                            /* a buffer for the returned value
                                           of Fetch_and_Store */

  Lock_q[NodeID] -> Successor = Nil;    /* initialize the successor */
  Pred = Fetch_and_Store(Lock, NodeID); /* fetch and then update the Lock */
  If (Pred != -1)         {             /* the Lock is not free */
     Lock_q[NodeID] -> Get_Lock = 0;    /* set myself NO Get_Lock */
     Lock_q[Pred] -> Successor = Lock_q[NodeID];  /* link myself to the end */
     While(Lock_q[NodeID] -> Get_Lock != 1);      /* local spin */
  }
}

/* processor NodeID releases the Lock */
Release_Lock (NodeID, Lock)
  Short NodeID;
  Short *Lock;
  {Int Pred;                            /* a buffer for the returned value
                                           of Fetch_and_Store */

  If (Lock_q[NodeID] -> Successor != Nil )
     Lock_q[NodeID] -> Successor -> Get_Lock = 1; /* wake up the successor */
  Else {
     Pred = Fetch_and_Store(Lock, -1);       /* set the lock free */
     If (NodeID != Pred) {                   /* some one gets in again */
        Pred = Fetch_and_Store(Lock, Pred);
     While(Lock_q[NodeID] -> Successor == Nil);  /* wait for releasing */
     Lock_q[NodeID] -> Successor -> Get_Lock = 1;
        }
     }
  }
```

Figure 8. Acquire and release functions of array-link-based distributed lock.

number of processors (the horizontal axis). Each processor requests lock acquisitions 1,000 times in a loop. We set the critical sections for all these experiments to 0. In addition, we ran all experiments on both the GP1000 and the TC2000 under benchmark mode (the systems did not process anything else during the experiments).

Figure 9 shows the performance of the spin-lock algorithms on the GP1000 with up to 120 processors. The poorest performing algorithms are Test_and_Set, Test_and_Set with a static delay, and Test_and_Set with an exponential delay. The delay functions do not improve the performance of Test_and_Set as effectively as on the Butterfly I,[2] because the GP1000 has higher network contention. The ticket-lock algorithm with proportional delay gives better performance, but it is unlikely to scale well because software global spin causes more network contention. The only scalable algorithm is the distributed lock.

Figure 10 gives performance results for the the TC2000. The simple Test_and_Set lock performs the poorest in scaling and time spent. The static-delay function slightly improves and the exponential-delay function significantly improves performance, but neither variation is scalable. The ticket-lock algorithm scales better than on the GP1000 because of network improvements for software global spin, but compared with the distributed-lock algorithm it will not scale to a very large number of processors. Nevertheless, its performance could be acceptable with a medium number of processors.

Earlier experiments found that the Test_and_Set lock with exponential backoff, the ticket lock with proportional backoff, and the MCS lock all scale very well on the Butterfly I.[2] (We expect that the MCS lock will scale on both the GP1000 and the TC2000.) Our results with the GP1000 and TC2000 have a slope for the ticket curve about seven times larger than for the distributed lock. Hence, it and all the Test_and_Set algorithms do not scale on these two machines. They are centralized algorithms with global spin implemented in two different ways, both of which cause network contention. On the other hand, the distributed lock should perform well even with thousands of processors competing on a MIN-based multiprocessor.

Performance on the KSR1

We ran the same experiments on KSR1 with two rings and a total of 64 processors. This small system is a reasonable testbed because it covers all three levels of cache access. The software implementations for Butterfly and KSR1 are quite different, but the principles of the algorithms and architectures are the same. (The KSR1's get-subpage algorithms correspond to the Butterfly's Test_and_Set.)

As shown in Figure 11, the worst performer is the ticket-lock algorithm with proportional delay. The algorithm does take advantage of a local spin while waiting to enter the critical section, but it suffers badly from overhead to maintain cache coherence (from the "Ping-Pong" effect mentioned earlier). The algorithm also requires extra time to implement its FIFO service. For example, in a one-level clockwise ring with 32 processors, suppose the sequence in which processors should enter the critical section is 0, 15, 6, 3, 30, and 29. After processor 0 exits the critical section, it updates the lock Current so the next processor can enter the critical section. To reach processor 15, the ring skips processors 6 and 3, which are physically closer to processor 0. After processor 15, processor 6 has its turn — and to reach it the ring skips processor 29 and 30 and *again* skips processor 3. The skipping continues to maintain FIFO service.

This example with only six processors in one ring suggests that such "skipping over" would create even longer delays with more processors. Comparing Figures 10, 12, and 13 shows that the ticket-lock algorithm with proportional delay is much less scalable on the KSR1 than on the BBN GP1000 and TC2000. In fact, on the KSR1 this algorithm performs very poorly.

The distributed-lock algorithm also guarantees FIFO service and scales poorly. Like the ticket-lock algorithm, this algorithm suffers from "Ping-Pong" overhead, although not as much because the globally shared array is distributed throughout the processors' caches. Each processor allocates a lock structure on which it spins locally. As each processor stores the address of its local lock structure to its appropriate element (the logical number of its p-thread) of the shared array, it has in its cache the only valid entry of that array element. In other words, this is the equivalent of a simple write operation.

Unlike the GP1000 or the TC2000, the KSR1 has no remote write operation. As a processor spins on its local structure (waiting to enter the critical section), it is "awakened" by its predecessor when it exits the critical section. Before the predecessor writes to update its successor's lock, it invalidates the successor's cache for its

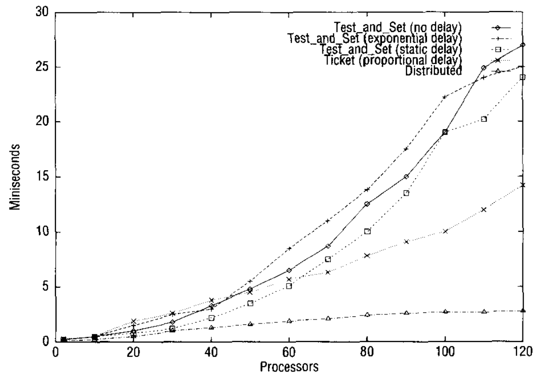


Figure 9. Performance of spin-locks on the GP1000.

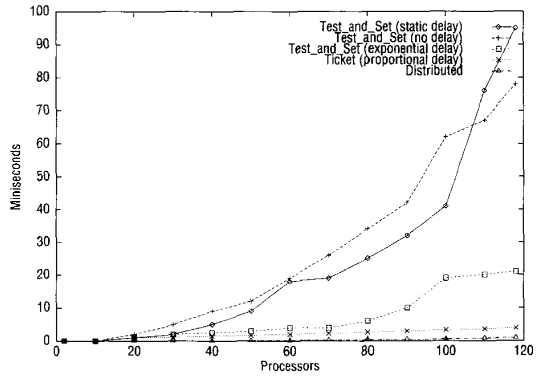


Figure 10. Performance of spin-locks on the TC2000.

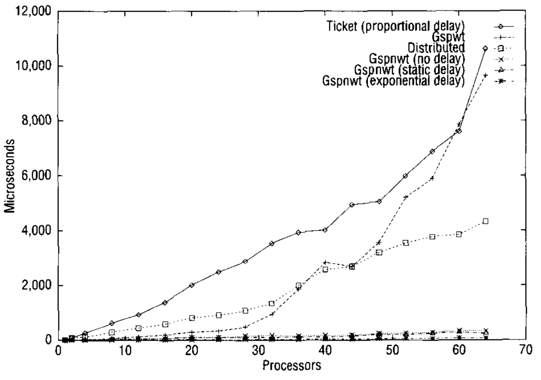


Figure 11. Performance of spin-locks on the KSR1.

local lock structure. Because there is no remote write in the KSR1, this is done inside the predecessor's cache. The successor will then have a read-miss on its local lock

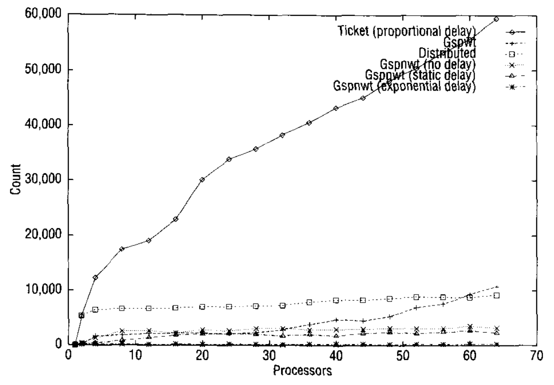


Figure 12. The number of cache-subpage misses of spin-locks on the KSR1.

while it is spinning, causing it to acquire the new value from the predecessor's cache.

This "Ping-Pong" effect does not occur as often as in the ticket-lock algorithm because the array element (and thus the local lock structure) of each processor is invalidated during the local spin only once while it is waiting to enter the critical section — when its predecessor wakes it up. In the ticket-lock algorithm, on the other hand, the lock is invalidated when *any* processor is awakened because it is globally shared by all processors. Hence, the distributed-lock algorithm performs better than the ticket lock, but its guarantee of FIFO service leads to the same processor skipping. As a result, unlike on the GP1000 or the TC2000, the distributed-lock algorithm does not scale on the KSR1.

The Gspwt (get-subpage, wait) algorithm falls between the ticket-lock and distributed-lock algorithms and does not scale. The failure is not caused by the "Ping-Pong" effect or guaranteed FIFO service (which it cannot implement, as we will show). Instead, performance degradation results because a processor stalls if it cannot get the lock subpage in an atomic state with one message.

The KSR1's architecture is a big factor in this algorithm's performance. In a one-level ring with 32 processors, suppose the sequence in which the processors enter the critical section should be 0, 30, 20, 11, 3, 15, and 4, since this is the order in which they put out a request for the subpage. As processor 0 obtains an atomic state on the lock and is in the critical section, other processors attempt to acquire the lock. As these requests go on the ring, the lock subpage in the holding cache enters the *transient-atomic* state. When processor 0 releases the subpage, the transient-atomic state forces the subpage onto the ring in a special release state, ready to be accepted by *any* processor previously stalled because it could not acquire the subpage in an atomic state. The subpage rotates clockwise on the ring, so the order in which the stalled processors enter the critical section is 3, 4, 11, 15, 20, and 30. This sequence does not match the sequence of requests, and some processors stall longer than they should, but eventually each processor gets the subpage. Performance degrades more as the number of processors increases, especially if the processors are on other rings.

The best algorithm in Figure 11 is Gspnwt with exponential delay, followed by Gspnwt with static delay and Gspnwt with no delay. The algorithm with no delay performs much better than the ticket, distributed, and Gspwt algorithms. Processors continuously try to acquire the subpage, placing requests on the ring until each has succeeded. These requests obviously burden the AllCache engines, but the approach gives each processor a better chance of acquiring the lock when the holding processor releases it. There is a "Ping-Pong" effect like that in the ticket-lock and distributed-lock algorithms, but there is very little chance of skipping processors and no stalling as each processor continues to try to acquire the lock. A processor may in fact be able to reacquire the lock an additional time if it puts out its next request fast enough (thus, there is no cache subpage miss). However, because all processors have an equal chance to acquire the lock, this may not occur as often when using static delay or exponential delay (explained below). The random way that the processors acquire the lock balances pretty fairly in terms of opportunity and length of time for lock acquisition. Of course, with more processors, time to acquire a lock increases.

Static delay improves the Gspnwt algorithm's performance. When a processor cannot gain the subpage, a busy-wait delay of its reattempt for a certain number of iterations relieves the burden on the AllCache engines. However, this approach does lessen fairness. Delay gives the processor holding the subpage an edge over the remaining processors: When the current processor sets the subpage of the lock to an atomic state, the remaining processors delay their next attempt. During this delay the current processor may release the subpage (especially if the critical section is very small). If it wants to reenter the critical section (iteration is set to 1,000), the same processor can reacquire the lock before the other processors reach the end of their delays. Eventually, another processor may sneak in and take over the monopoly for a while. This phenomenon improves timings when they are averaged, since a processor accessing from its own local cache takes less time than if it goes to the ring.

Exponential delay brings even greater improvement. Requests on the AllCache engines and fairness in

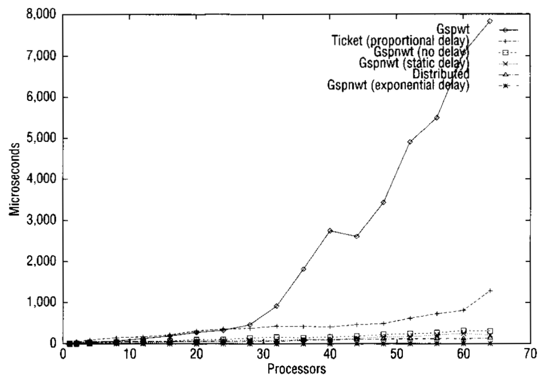


Figure 13. The time of cache-subpage misses of spin-locks on the KSR1.

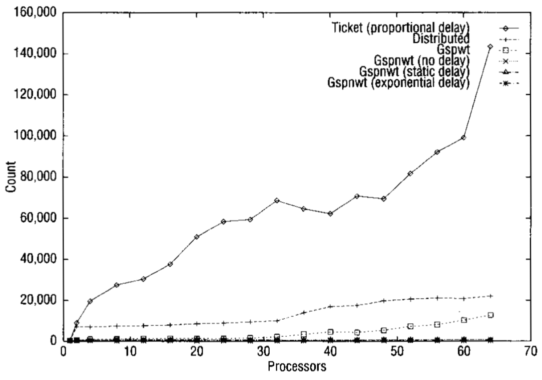


Figure 14. The data subcache subblock misses of spin-locks on the KSR1.

processor access are less than in the static-delay algorithm, because delay increases exponentially if a processor cannot acquire the subpage. Now a very big advantage goes to the processor with the lock. Since it can gain access to the lock again and again while the other processors are delaying (and the delay is growing exponentially), its timing is very small. A processor may be able to enter and exit the critical section up to several hundred times before another processor sneaks in and gets the lock for itself. Timings are good, since all of those accesses are done from the local cache. Even though most processors wait before acquiring the lock for the first time, once a processor has its subpage in an atomic state, the remaining timings lower its average. Figure 11 shows that the Gspnwt with exponential delay does not scale perfectly. Times increase as more processors try to enter the critical section, but on the KSR1 this algorithm is overwhelmingly the best.

If the critical section is small and the order in which processors gain access to it is not very important, then Gspnwt with exponential delay is clearly the best choice. If, however, fairness in processor access is important, then Gspnwt with no delay is the most appropriate. If requirements are somewhere between the two, then Gspnwt with static delay is the choice.

Figures 12-15 use different statistics to show the superiority of the get-subpage algorithms on the KSR1. Figure 12 shows that the ticket-lock algorithm has the most cache-subpage misses. Subpage misses (because of the "Ping-Pong" effect) are not as high in the distributed-lock algorithm. The Gspwt algorithm also has many subpage misses because when one processor has acquired the subpage, it is counted as a subpage miss in the other processors and forces them to stall. The three Gspnwt algorithms have fewer subpage misses because once a processor has the subpage, it may reacquire it immediately (hence no subpage miss). (This is especially true with static or exponentially growing delays.)

Figure 13 shows the time it takes to determine that a subpage is not in the local cache, to send a request for that subpage through the AllCache engines, and to receive a copy of the referenced subpage in the local cache. The Gspwt algorithm is the worst performer because processors stall waiting for the subpage to come around the ring. Second worst is the ticket-lock algorithm because of its "Ping-Pong" of invalidations among the processors' local caches. Gspnwt with no delay and with static delay performed well because of the processors' ability to reacquire the subpage immediately from within their own local cache. The distributed-lock algorithm also performed well because of its local spin and weaker "Ping-Pong" effect. The best was Gspnwt with exponential delay. It offers processors the greatest opportunity for in-cache lock reacquisition.

Whenever a processor uses data, it places the data in its local cache *and* its data subcache. The most recently used data will be in the data subcache. If a processor needs to reference the same data, it first checks the subcache to see if the data is still there. If not, then it checks its local cache. If the data is not there, it puts a request on the ring. Figure 14 shows misses in the data subcache. The relations among the different algorithms' plots are similar to those for cache-subpage misses (Figure 12). The worst performer is the ticket-lock algorithm because of the many invalidations in the local spin. The distributed-lock algorithm suffers less because invalidation goes only to a processor's successor. The Gspwt algorithm performs slightly better than the distributed-lock algorithm but suffers because one processor's acquisition of the subpage causes a data subcache miss (and a cache-subpage miss) in all other processors. All three

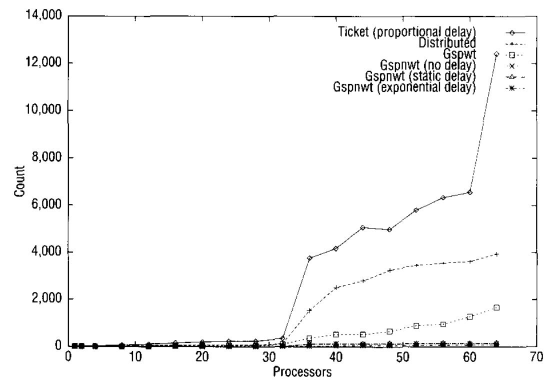


Figure 15. The number of packets traveling through AllCache engine 1 for spin-locks on the KSR1.

Gspnwt algorithms perform well for the same reasons they perform well in cache-subpage misses. The local caches where processors can reacquire the lock are really their data subcaches.

Figure 15 shows the number of packets that travel through AllCache engine 1. The ticket-lock algorithm performs the worst because its many invalidations require many packets. The distributed-lock algorithm uses somewhat fewer packets because it reduces this "Ping-Pong" effect. The Gspwt algorithm requires more processor requests (and hence packets) than the Gspnwt algorithms. With the Gspnwt algorithms, a processor may send many packets on the ring to acquire the subpage, but then it can reacquire the subpage locally, keeping the overall packet count low.

In both MIN-based and HR-based systems, processors can access shared memory or cache space at different distances with different timing costs. Butterfly systems have a local/remote memory-access model, while the KSR1's model has local, level 0 ring, and level 1 ring cache access. Hence, both architectures use nonuniform memory access (NUMA). A problem with NUMA parallel processing is that for good performance a user must explicitly manage processor locality, use of processor resources, and program execution. This can cause serious performance degradation on MIN-based multiprocessor systems,[6–8] but very little work has been done to evaluate the problem on the KSR1.

Our experience indicates that the HR-based KSR1 handles potential contentions more efficiently than the MIN-based BBN GP1000 and TC2000. There are several reasons for this. In the KSR1 hardware, high-speed search engines supported by processor directories and routers handle directory searches and data movement, reducing remote-cache access time and the cost for cache invalidations. Also, the KSR1 architecture has higher bandwidth links, more efficient broadcast, and better ordering properties. Unlike a MIN-based processor, a KSR1 processor has a direct communication link to only one processor node (its neighbor on the ring). A processor loads all messages to the slotted ring for delivery to target nodes. The rotating ring orders and delays remote data-access requests, naturally reducing network contention for programs with hot spots.

An HR-based architecture exploits hierarchical locality of reference by moving referenced data to a local cache and satisfying data references with nearby copies whenever possible. However, on the KSR1, parallel-processing performance still suffers from the NUMA problem. As our spin-lock synchronization study shows, large numbers of cache-access misses and inefficient cache use significantly degrade performance.

The next wave of massively parallel computers will probably make programming easier by giving the illusion of shared memory. However, this illusion might hide not just the distributed space, but also processor and memory localities that would improve parallel performance. Such systems may not perform as well as they might unless there are breakthroughs in compiler and architecture techniques that automatically exploit localities.

Acknowledgments

We thank Neal Wagner and George Butchee for carefully reading the article and contributing many valuable suggestions and comments. This work has been supported in part by the National Science Foundation under research grants CCR-9008991 and CCR-9102854, and under instrumentation grant DUE 9250265. It has also been supported by the US Air Force under research agreement FD-204092-64157, a Cray Research grant, and by a fellowship from the Southwestern Bell Foundation. Experiments were conducted on the BBN GP1000 at BBN Advanced Computer, Inc., the BBN TC2000 at Lawrence Livermore National Laboratory, and the KSR1 machines at Cornell University and the University of Washington.

References

1. T.E. Anderson, "The Performance of Spin-Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 1, Jan. 1990, pp. 6–16.
2. J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems*, Vol. 9, No. 1, Feb. 1991, pp. 21–65.
3. D.P. Reed and R.K. Kanodia, "Synchronization with Event-Counts and Sequencers," *Comm. ACM*, Vol. 22, No. 2, Feb. 1979, pp. 115–123.
4. E.W. Chan, *Comparative Performance Evaluation of Synchronization Alternatives Between MIN-Based and HR-Based Shared-Memory Multiprocessors*, master's thesis, University of Texas, San Antonio, 1993.

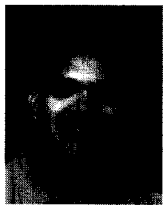
5. X. Zhang and W. Wu, "On Measuring and Evaluating Synchronization and Virtual Memory Performance of a Multiprocessor with Multistage Interconnection Network," *Proc. 25th Hawaii Int'l Conf. Systems Sciences*, Vol. 1, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 218–224.

6. R.P. LaRowe, Jr., and C.S. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Trans. Computer Systems*, Vol. 9, No. 4, Nov. 1991, pp. 319–363.

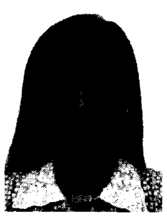
7. X. Zhang, "Dynamic and Static Load Balancing for Solving Block Bordered Circuit Equations on Multiprocessors," *IEEE Trans. Computer Aided Design*, Vol. 11, No. 9, Sept. 1992, pp. 1086–1094.

8. X. Zhang and X. Qin, "Performance Prediction and Evaluation of Parallel Processing on a NUMA Multiprocessor," *IEEE Trans. Software Eng.*, Vol. 17, No. 10, Oct. 1991, pp. 1059–1068.

Xiaodong Zhang is an associate professor of computer science and director of the High-Performance Computing and Software Laboratory at the University of Texas at San Antonio. He has also been a visiting faculty member at Rice University and Texas A&M University. His research interests are parallel and distributed computation, parallel system performance evaluation, and numerical analysis for solving nonlinear equations and optimization problems. He received his PhD and MS in computer science from the University of Colorado at Boulder in 1989 and 1985, and his BS in electrical engineering from Beijing Polytechnic University in 1982. He is a member of ACM, the IEEE Computer Society, and SIAM.



Robert Castañeda is a graduate student in computer science at the University of Texas at San Antonio. He is a recipient of a 1993 Southwestern Bell Foundation graduate fellowship, and of the 1994 University Life Award for academic performance. His research interest is performance evaluation of parallel and distributed architectures and systems. He received his BS in computer science from the University of Texas at San Antonio in 1990.



Elisa W. Chan is a product manager at Emis Software in San Antonio, Texas, where she is concerned with software development. She received her MS in computer science from the University of Texas at San Antonio in 1993, and her BS in computer science from the Beijing Computer Institute in 1987.

Zhang and Castañeda can be reached at the High-Performance Computing and Software Laboratory, University of Texas, San Antonio, TX 78249; Internet zhang@ringer.cs.utsa.edu. Chan can be reached at Emis Software, 901 N.E. Loop 410, San Antonio, TX 78209; Internet wchan@dragon.cs.utsa.edu