Efficient Synchronization Algorithms Using Fetch & Add on Multiple Bitfield Integers

by

Eric Freudenthal
Olivier Peze

Ultracomputer Note #148
February, 1988

This work was carried out as part of the research program on parallel processing at the Ultracomputer Research Laboratory of the Courant Institute of Mathematical Sciences. It was supported under a Joint Study Agreement with IBM, grant number IBM-N0039-84-R-0605.

Current Address for Olivier Peze: Ecole Nationale Superieure des Telecommunications de Paris, France.

ABSTRACT

Efficient interprocess synchronization tools for MIMD shared memory computers are critically important for the success of such systems. As members of the Ultracomputer Project at NYU, the authors have been participating in the continuing development of such algorithms. From their work, partially represented in this paper, they have observed that many of the more efficient algorithms utilize the storage and manipulation of more than one state variable in a single machine integer using fetch-and-add (see [GLR 83]). This paper describes this technique and presents several new implementations of busy-waiting synchronization functions which utilize this facility.

1. Introduction

The NYU Ultracomputer is a shared memory MIMD computer (see [Gottlieb 86]). Each processor has a local cache for private data and code, and is provided access to an interleaved shared memory via a logarithmic multistage interconnection network. To avoid contention in the memory interface network, the ultracomputer combines concurrent load, store, and fetch-and-add [2] requests to the same memory location.

Unfortunately, due to the requisite network traversal, shared memory MIMD computers such as the NYU Ultracomputer and the IBM RP3 suffer from high latency on references to shared memory. The algorithms presented in this paper store several variables worth of state information in a single machine word. These state variables can be read and modified by a single network reference (faa), providing significant time savings over other algorithms which need to determine state information in several accesses amd sometimes requiring additional locking mechanisms. In addition, the ability to fetch and modify several state variables atomically is very convenient in the development of synchronization algorithms.

Harrison (see [Harrison 86, 88]) has proposed the use of combining positive bitfield increments to special *add* and λ processors which would have been substituted for memory. These *add* and λ processors would directly support synchronizations primitives. The value returned by an *add* and λ operation would indicate the state of some lock rather than the actual value of a memory cell modified by fetch-and-add. In contrast, the algorithms presented in this paper do not utilize any special hardware except fetch-and-add.

The views and opinions expressed herein do not necessarily reflect those of IBM.

[1] Fetch-and-add (faa) is a hardware primitive which allows any number of processors to read (fetch) the value of an integer and add an increment to it atomically. For example, the instruction $faa(\&i,3)$ increments the value of *i* by three, and returns the value of *i* previous to the increment. On the NYU Ultracomputer, and the IBM RP3, concurrent fetch-and-add requests to the same memory address are combined in the processor-to-memory network, yielding results which are equivalent to some serialization without serialization delays. (see [Gottlieb 86])

2. Fetch-and-add on bitfield variables

Bitfield variable storage, sometimes called *packing*, is a method of storing more than one variable in a single machine word. This mechanism is often used in serial programs to conserve space. The algorithms presented in this paper will be using an extension of this technique which supports parallel access to such variables.

Current machines support 32 bit words and macros (to be described below) hide bit packing order. However without loss of generality, for the following discussion, we shall assume that a machine word is eight bits wide, and that a *C* compiler allocates bitfield variables (see [K&R 78]) most significant bit first.

For example, the following code, in the language **C** defines the variable *x* which represents a single eight bit machine word.

```
struct { /* assume compiler assigns msb first */
        unsigned 4:b;
        unsigned 4:a;
} x;
```

Which is represented as follows:

Fields a and b in a Eight Bit Machine Word								
field name:	b				a			
bit location:	7	6	5	4	3	2	1	0
b = 3; a = 2:	0	0	1	1	0	0	1	0

The least significant four bits, from bit zero to three, are used to store the integer field *x.a*. The next four bits, from bit four to seven, are used to store the other four bit integer field *x.b*. Of course, these variables are of restricted range, they can not become negative, nor can they have a value which requires more bits to represent than has been allocated. Both *x.a* and *x.b* are four bit wide fields, which restricts their values to the range of zero to 15. (which is $2^4 - 1$).

Serial programs reference these fields by shifting and masking. Unfortunately, such references must occur serially, making them unusable for shared access in a parallel shared-memory MIMD environment.

Bitfield variables can be modified by addition. Observe that since the field *x.a* is stored in the least significant bits of the machine word, fetch-and-add (*faa*) can directly modify this field. Modifying the other field of a is a bit more tricky. To add δ to *x.b*, δ must be multiplied by sixteen (which is equal to 2^4). For example, the operation $faa(\&x, 3*16)$ will add three to the field *x.b*.

Fortunately, fetch-and-add accesses to bitfield variables can be combined. For example: The instruction $faa(\&x, delta)$ increments the $x.a$ field by δ as long as the value of the $x.a$ field stays within its restricted range. If this restriction was not followed, the resulting carry or borrow would disturb the value of other field in the same word of storage. In addition, a *snapshot* before the increment of the entire variable (including both of its fields) is returned by the fetch-and-add.

A fundamental feature of fetch-and-add on bitfield variables is that more than one field can be modified by a single addition if they are all stored in the same machine word. For example, the instruction $faa(\&x, i+j*16)$ increments $x.a$ by i, and $x.b$ by j atomically as well as returning the value of x prior to this pair of increments.

Both for efficiency and portability, macros have been implemented for the NYU Ultracomputer parallel **C** language so that bitfield variables can be conveniently referenced by fetch-and-add. The general form of these macros is **faa***n*$(var, field_1, \delta_1 [, field_n, \delta_n])$. For example, the macro $faa2(x, a, i, b, j)$ is equivalent to $faa(\&x, i+j*16)$ as explained in the previous paragraph.

3. Reader-reader lock

A reader-reader lock is a mutual exclusion algorithm where processes are members of one of two groups *A* and *B*. (see [Edler 84]) There is no limit to the number of processes (up to some predetermined maximum) in either group which can enter the critical section at one time. However only members of one of the groups can enter the critical section at a time. In many algorithms, a reader-reader lock can replace a binary semaphore allowing a higher degree of parallelism. Several applications will be discussed at the end of this paper. The original reader-reader lock was developed by Jan Edler at NYU and is used in the operating system and user-mode support code for the NYU ultracomputer. An algorithm similar to one presented in this paper was developed concurrently by Jerome Chiabaut at NYU.

3.1. The Algorithm

This implementation described below assigns the *A* lock higher priority than the *B* lock. Although this approach requires very few network accesses, it can lead to starvation of the lower priority lock. The end of this section mentions some variations on this algorithm one of which does not have this problem.

3.1.1. Variable

```
typedef struct {  /* 32 bit integer storage */
        unsigned aCount:16;
        unsigned bCount:16;
} rrLockT;
rrLockT abSem;
```

A long, shared integer, *abSem* is used to store all values. It will hold two bitfield values: *abSem.aCount* and *abSem.bCount*. *AbSem.aCount* holds the count of posted (possibly granted) *A* locks,

abSem.bCount holds the maximum count of posted *B* locks. Both the *bCount* and *aCount* fields are initialized to zero, indicating that neither lock is granted. Naturally, the number of processes which can request this lock must be restricted to the maximum value which can be stored in either field (in this case 65535).

3.1.2. Lower Priority Lock Implementation

```
/* *
   lock 'B' without priority over lock 'A'
* */
subroutine rrLockB(lock)
rrLockT *lock;
{
  rrLockT t;
 top:
  t = faa1(lock, bCount, 1);    /* posting -- increment bCount */
  if (t.aCount != 0) {          /* can't grant if 'A' lock granted or posted */
    t = faa1(lock, bCount, -1); /* decrement bCount */
    while (t.aCount != 0)       /* wait until no 'A' locks posted */
      t = *lock;
    goto top;                   /* try again */
  }
                                /* (granted) */
}
```

When a lock is requested, it atomically increments its count (**posting**) while checking that the other count is still zero. If so, then the lock is **granted**. If the other count is non-zero at the time that its own count was incremented, its count is decremented (to indicate that it does not hold a lock), and then *waits* until the other count's value is zero due to the release of the other lock. When this occurs, the waiting process restarts the requesting procedure again. If there is no contention for the lock, this routine requires one shared memory access.

3.1.3. Higher Priority Lock Request Implementation

```
/* *
   lock 'A' with priority -- locks out further 'B' requests until released
* */
rrPriLockA(lock)
rrLockT *lock;
{
```

```
  rrLockT t;
  t = faa1(lock, aCount, 1);    /* post by incrementing aCount */
  while (t.bCount != 0)         /* wait for 'B's to exit */
    t = *lock;
                                /* (granted) */
}
```

When a priority lock is requested, the requesting process first **posts** its request by incrementing its count and then waits until all locks on the other lock are released before **granting** the lock. By waiting without decrementing its count, this lock prevents further requests for the other lock from succeeding. If there is no contention for the lock, this function requires one network access.

3.1.4. Non-waiting (lower priority) lock request implementation

```
/* *
   request lock 'B' without priority
   returns 1 (true) if succeeded, 0 otherwise
* */
rrReqB(lock)
rrLockT *lock;
{
  rrLockT t;
  if (lock->aCount != 0)        /* don't increment if unavailable */
    return 0;                   /* not available, just return 'failed' */
  else {
    t = faa1(lock, bCount, 1);  /* only post once */
    if (t.aCount != 0) {        /* if unavailable... */
      faa1(lock, bCount, -1);   /* ...just decrement */
      return 0;                 /* return 'failed' */
    } else                      /* lock acquired */
      return 1;                 /* return 'succeeded' */
  }
}
```

This routine requests a lock, but doesn't wait if there is contention. It is useful in applications where there is other work which can be performed if the lock is unavailable. If there is no contention for the lock, this function requires two network accesses.

3.1.5. Releasing of locks

When a lock is released, its corresponding count is decremented. This function requires one network access.

3.1.6. Changing active lock types

```
/* *
   equiv to unlockB(), priLockA()...but atomic
   saves 1 shared access
* */
rrChangeBtoA(lock)
rrLockT *lock;
{
  rrLockT t;
  t = faa2(lock, aCount, +1, bCount, -1); /* remove myself from b lock count */
                                          /* and add myself to a lock count*/
  if (t.bCount != 1)            /* does anybody else hold a 'b' lock  */
    while (lock.bCount) ;       /* wait until all count b exit */
}
```

An already granted lock can lock out further requests to the same lock so that it can obtain the other lock by atomically incrementing the count for the other lock and decrementing its own count. When all the locks from its own group are released, then the lock for the other group is granted. This mechanism is used by a fair version of reader-reader lock which guarantees that neither lock type is starved.

If priority locking is being utilized, a lower priority lock can be changed to a higher priority lock, but not the reverse, otherwise deadlock could result. If both locks are requested by the symmetrical (non-priority) algorithm, then the either active lock type can be changed. If there are no outstanding lower priority locks, this function requires one network access.

3.2. The priority reader-reader lock is correct.

The priority algorithm utilizes lower priority requests for one lock type, and higher priority requests for the other.

For this algorithm to be correct, it must both ensure mutual exclusion between the two locks, and it must neither deadlock nor livelock, which would prevent either lock from being granted.

- **Neither count can become negative.**

  The counts initially are set to zero. The count for either lock is only modified by processes requesting or releasing it. They are always incremented before they are decremented. The number of

decrements is always less than or equal to the number of increments.

•If a process holds a lock, its count is greater than zero.

By above, no process will decrement a lock count without first incrementing it. If a lock is granted, the call which requested it incremented the count one more time than it decremented the count. Therefore the value of the count must be greater than zero.

• Both locks can not be granted at once.

The requested lock's count is incremented and the other lock's count is tested in an atomic *faa*. By the serialization principle of faa (see [GLR 83]), all such requests, regardless of lock type, must yield the same results as some serialization. Therefore, if a lock is already granted by one such operation, subsequent increment-test (*faa*) operations for the other lock will see the non-zero count, and not grant the lock.

• The priority algorithm can not lock-up indefinitely.

Livelock is a scenario equivalent to the degenerate naive-p (see [GLR 83]) where, due to some non-deterministic timing, no process can acquire a lock for a prolonged (perhaps infinite) period of time. Livelock differs from deadlock in that it is possible, in some serialization, for a process to be granted a lock in a livelock condition, but not in a deadlock condition.

We shall assume that there are a bounded number of processes (less than the maximum described previously) which are requesting locks, that no process ever attempts to hold two locks at the same time, and that every granted lock will be released in finite time. What needs to be shown is that independent of the timing of lock requests a lock must always be granted in a finite amount of time.

It is trivial to show that if only one type of lock is requested, it will be granted. Since only the count corresponding to the lock type which is requested is incremented, the count for the other lock will eventually become zero when all (if any) locks of the other type are released.

It is a little less obvious why it is impossible for this algorithm not to grant a lock of either type for an infinite amount of time. This can only occur if there are outstanding lock requests for both types of locks. In this case, the higher priority lock type will be granted to all outstanding requests for that type. This is because, after some finite amount of time, all processes requesting the lower priority lock, having detected the contention indicated by the non-zero other count, will have decremented themselves out of their own count, leaving it zero. At this time, the higher priority lock requests will be granted. If a higher degree of contention is expected, the lower priority lock request algorithm can be changed so that it waits for the higher priority lock's count to become zero before it performs the first decrement, reducing the likelihood of a prolonged delay before the higher priority lock is granted.

3.2.1. Symmetrical (no priority) algorithm can livelock.

It is necessary (although not obvious) that one of the lock types have priority over the others. Otherwise, if the higher priority lock code is modified so that it is symmetrical to the lower priority lock, there is a danger of livelock. This is because it is possible for the counts to be non-zero at a time that no locks are granted. When there are at least two concurrent unsatisfied lock requests for each lock type, the degenerate condition can occur when one of the counts is zero, the other non-zero, and no locks are granted.

A process requesting the corresponding lock to the non-zero count (assume *A*) will escape the while loop, and attempt to increment its variable. Before it increments its variable, the other count (*B*) is incremented (to non-zero) by a process which previously passed the initial while loop's test. The increment-test for the *A* requests and the *B* requests both fail. When either of them decrements its count to zero, additional requests for the other lock are allowed to exit the while loop and start their own increment-test/decrement posting sequence. In this manner, requests for each lock can exit the while loop when no locks are granted, even though the counts are non-zero.

State	Action	A Count	B Count	Holds Lock
1	A_1,A_2,B_2 pass while test	0	*0*	nobody
2	A_1++:succeeds; granted	A_1	*0*	A_1
3	B_2++:fails	A_1	B_2	A_1
4	A_1--(release)	0	B_2	nobody
5	A_1 exits	0	B_2	nobody
6	B_1 passes while test	0	B_2	nobody
7	A_2++:fails	A_2	B_2	nobody
8	B_2--:loops	A_2	*0*	nobody
9	A_1 passes while test	A_2	*0*	nobody
10	B_1++:fails	A_2	B_1	nobody
11	A_2--:loops	0	B_1	nobody
12	B_2 passes while test	0	B_1	nobody
13	A_1++:fails	A_1	B_1	nobody
14	B_1--:loops	A_1	*0*	nobody
15	b passes while test	A_1	*0*	nobody
16	B_2++:fails	A_1	B_2	nobody
17	A_1--:loops	0	B_2	nobody
6	B_1 passes while test	0	B_2	nobody

3.3. Other flavors of related reader-reader locks

A symmetrical algorithm, without priority, can be realized by adding a pair of boolean fields, one to indicate if each lock is granted, into the bitfield integer which indicates if any process actually holds either lock. The initial test is modified so that requests for a particular lock will be posted iff both the other count is non-zero (as before) and if the count for locks of the same type is non-zero or a lock of the same type is granted (as indicated by the corresponding boolean flag).

An asymmetric algorithm which guarantees that neither lock type can be starved by a continuous stream of either type of lock request can also be realized by extending the priority lock. This *fair* algorithm counts unsatisfied requests, and forces the lock to regularly change phase if there are unsatisfied requests.

4. Reader-writer lock

A readers-writers lock ([GLR 83]) is a synchronization algorithm which protects a critical section of code. Processes are broken into two groups, *readers* and *writers*. Any number or *reader* processes are allowed to hold the lock concurrently, but only one *writer* process is allowed to hold a lock at a time. Of

course, reader and writer locks are mutually exclusive. A simple priority reader-reader lock which only allows a single reader from the higher priority group to enter the critical section. This algorithm is equivalent to the one presented in [GLR 83], however that algorithm did not formalize the notion of bitfield variables and their general usefulness.

```
typedef rrLockT rwLockT;
#define rwLockReader(lock) rrLockB(lock)
#define rwUnlockReader(lock) rrUnlockB(lock)
#define rwUnlockWriter(lock) rrUnlockA(lock)

rrLockWriter(lock)
rwLockT *lock;
{
  abCountT phase;
 top:
  phase = faa1(&lock->abCount, aCount, 1); /* post by incrementing aCount */
  if (phase.aCount != 0) { /* are we first? */
    phase = faa(&lock->abCount, aCount, -1); /* no; decrement, wait till 0 */
    for (phase.aCount -= 1; phase.aCount != 0; phase = lock->abCount);
    goto top;
  } else                   /* we are first */
    while (phase.bCount != 0) phase = lock->abCount; /* wait for Bs to exit */
}
```

5. Wave and Phase Synchronization

Many algorithms are synchronous in nature, having sections which need to work in lock-step with varying granularity. In such algorithms, a critical section extends through several sychronization points where all processes in the critical section must arrive before they can continue.

5.1. Barrier synchronization

Barrier synchronization provides a mechanism to synchronize a group of processes which need to step through a problem together. When a program calls *barrier()* it is delayed until all of the processes with which it is synchronizing also call *barrier*.

The data structure is a bitfield integer with two fields:

```
typedef struct {
  unsigned 15:P;               /* # processes */
  unsigned 17:B;               /* barrier count, see range()  */
```

```
} barrierT;
```

Which is stored as follows:

Field Allocation for 32 Bit Barrier		
Name	P	B
Usage	number of processes	barrier count
Useful range	maximum number of processes	0..3*P
Field width	15 bits	17 bits
Bit locations	31..17	16..0

The *P* field is initialized to the number of processes which will be synchronizing, and the *B* field is initialized to zero.

```
/* *
   range returns true if B is in range P..2 * P -1
   normally coded as a macro; shown as a function for clarity
* */
range(b)
barrierT b;
{
  return(b.B >= b.P && b.B < b.P * 2);
}

/* *
   barrier() wait until 'P' processes call 'barrier'.
   based on Dimitrovsky barrier algorithm
   resets for next barrier automatically
* */
barrier(bar)
barrierT *bar;
{
  barrierT t;
  int oldRange;
  t = faa1(bar, B, 1);          /* register myself at barrier */
  oldRange = range(t);          /* note the range before I registered */
```

```
  t.B = t.B + 1;                /* adjust B to value after I registered */
  if (oldRange == range(t)) {   /* am I last to register? */
    if (oldRange == 1)          /* did I increment from phase 1->0 */
      faa1(bar, B, -2 * t.P);   /* yes, reset to low of range 0 */
   } else                       /* I was not last to register */
     while (oldRange == range(*bar)) ; /* wait for others to register */
}
```

The algorithm works by incrementing the *B* field through two ranges: range 0 includes values from 0 to $P-1$ and range 1 includes values from P to $2*P-1$. Values greater than or equal to $2*P$ are considered as part of range 0. At each sync point, each process calls *barSync* which uses fetch-and-add to increment *B* by one, noting the range which *B* was previously in. Then all processes wait until *B* is incremented to the next range. When a process whose increment causes the transition from range 1 to 0 (incrementing from $2*P-1$ to $2*P$, it will decrement *B* by $2*P$.

Observe value of the *B* field is bounded by $3*P$ for the following reason: At synchronization points where *B* field started in range zero, all processes must wait until *B* is incremented to the value $2*P$. While one of these processes is decrementing *B* by the same amount, the other processes may continue on to the next barrier point. Assume that all of the other processes reach the next barrier point before *B* is decremented. If the slow process is a member of the group of processes which will be synchronizing at the next barrier, only $P-1$ processes are available to increment *B*, restricting the value of *B* to less than $3*P-1$. If this process is not a member of that group, then the limit is 3*P. Because of its range, two more bits are allocated to *B* than *P*, causing the maximum number of processes which can use this implementation (with 32 bit words) to be restricted to $2^{15}-1$ which is 32767.

The cost of a call to *barrier*( ) is at least one network access for one process in the group (on the transition from range 0 to range 1), and at least two for the others. Dimitrovsky's original implementation required two or three accesses because the number of processes was kept in a separate variable. Admittedly, this savings is small, and since the value of *P* is invariant over the critical section, it could be copied into private memory. This implementation also differs from that presented in [Dimitrovsky 85] in that, he did not consider values of *B* greater than $2*P-1$ to be in range 0, therefore all of the processes needed to wait until after *B* was reset to zero, causing at least one extra delay due to network references for most processes.

5.1.1. Two Phase Lock

A simple two phase barrier synchronization algorithm can be achieved utilizing the priority reader-reader lock presented above. The first phase is protected by lock *B*, and the second phase is protected by lock *A*. Since these two locks are exclusive, there is no risk of processes being in both phases at the same time. Entry to the first phase is granted by a call to *lockB*( ), the transition to phase two is achieved by calling *changeBtoA*( ) which locks out further requests for the first phase and locks the second phase. The

lock is released by calling *unlockB.* If the *fair* reader-reader lock mentioned above is used instead of the *priority* lock, this lock will also have the wave property. A feature of this algorithm is that the number of processes which are synchronizing together is determined automatically.

Implementaton of Two Phase Lock			
function	*what it does*	*how implemented*	*min cost in network refs*
twoPhaseLock0	enter first phase	rrLockB	1
twoPhaseLock0to1	enter second phase from first	rrChangeBtoA	1
twoPhaseUnlock1	exit second phase	rrUnlockA	1
twoPhaseUnlock0	exit first phase without entering second	rrUnlockB	1

5.2. Wave Synchronization

Wave synchronization (see [Dimitrovsky 86]) breaks processes requesting entry to a critical section into groups such that a process is only allowed to enter the critical section no more than one time for each group. This smaller group can synchronize at barrier points inside the critical section. Furthermore, a process which is waiting for entry to such a critical section only must wait until the current group releases all of its locks before it can enter with the next group. Note that the two phase barrier algorithm has the wave property.

5.2.1. Group Lock

Group lock is a barrier synchronization algorithm (see [Dimitrovsky 86]) which is useful for algorithms where a group of processes need to synchronize at one or more barriers and the size of the group is unknown. It also has the *wave* property. It consists of three subroutines: *gLock*, *gSync*, and *gUnlock*. Typical usage is as follows:

```
gLock()
... code for phase 0
gSync()
... code for phase 1
                .
                .
                .
gSync()
... code for phase n
gUnlock()
```

The effect of group lock in the above example is to make sure that processes are only executing code in one phase at a time. Processes which call *gLock()* when other processes hold active locks must

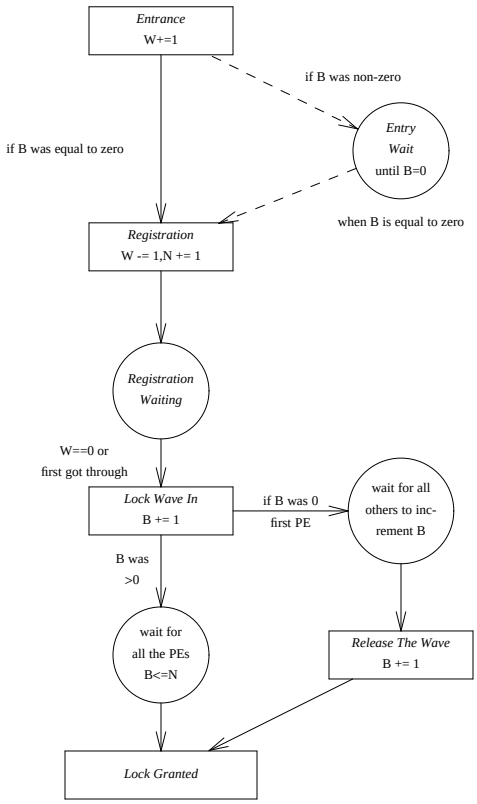
wait until the glock is next available. This algorithm (as does the original group-lock implementation) guarantees that a process which must wait for the lock to be released will be granted the lock with the next group.

Several implementations of group lock using bitfield variables have been developed. The following implementation, utilizes only thirty-two bits of storage, limiting it to $2^{10}-1$ processes. Less elegant implementations have been developed which require more storage but allow a much larger number of processes to use the lock.

This algorithm requires three bitfields as follows:

Field Allocation for 32 Bit Group Lock			
Name	W	N	B
Usage	# of waiting processes	# of processes in critical section	barrier count
Useful range	$0..2^{10}-1$	$0..2^{10}-1$	$0..3*N$
Field width	10	10	12
Bit locations	31..22	21..12	11..0

The lock algorithm is outlined in the flowchart below; operations which modify the synchronization variable and must occur atomically using a single fetch-and-add are shown inside boxes and comments are italicized.



The lock works as follows. Processes which would like to enter the critical section first increment the waiting count *W* and test the atomically returned value of the barrier count *B* to determine whether *B* was zero, indicating that this process can enter the lock with the current wave. A non-zero value of *B* would have indicated that there are already processes inside the lock, requiring the process to wait until the next group.

Once a process has determined that it will enter the lock, the first step is to atomically increment *N* and decrement *W* by one. Note that this operation does not affect the sum $W+N$.

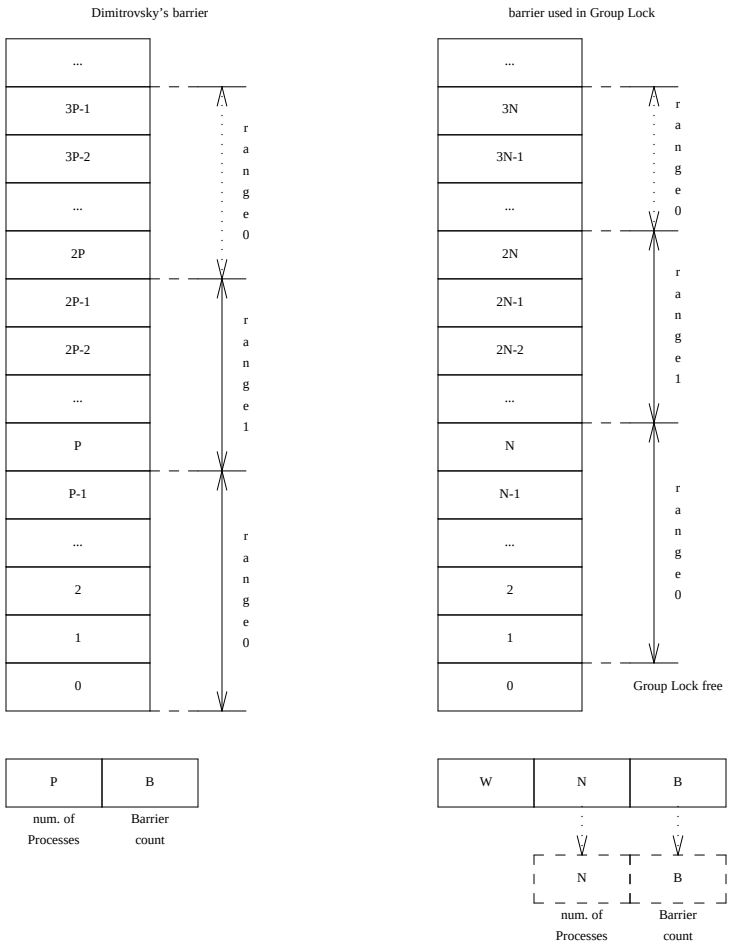
In order to ensure that all processes which were required to wait until the previous group released the lock, all of the processes must wait until at least one of them sees the value of *W* decremented to zero, indicating that there are no processes at the entry waiting step.

Once a process (possibly several) notices that *W* has a zero value, it increments *B*, noting the values of all three fields at the time of the increment. Note that the first of these increments makes *B* non-zero, causing processes which subsequently try to enter (by incrementing *W*) to wait until the next group is allowed to enter the first phase of the critical section. The non-zero value of *B* also ensures that the

processes in the *regisitration waiting* will continue and increment *B*. The first process to increment *B* knows the total number of processes in this group, which is $W+N$. Also observe that until this process performs a second increment on *B*, $B \leq N$.

Once all of the processes have *locked the wave* by incrementing *B*, and wait until $B = N = actual\ number\ of\ processes\ in\ this\ wave$. All of these processes except the first now wait for *B* to take on a value greater than *N*. The first process notices that all of the others have incremented *B* and performs the extra increment, allowing all of the processes to enter the lock.

The function *gBarrier*( ) works analogously to the function *barrier*( ) described above with the definition of the ranges modified as shown below.



The unlock function works by decrementing *B*. The last process to decrement *B* (as evidenced by $B=1$ or $B=N+1$) performs an additional subtraction of both the value of *B* and *N*. This function requires either one or two network accesses. A slightly more expensive alternative unlock function can also be constructed which allows some processes to leave the critical section between synchronization points.

This algorithm requires at least seven network accesses for a lock/barrier/unlock sequence. This compares favorably with the minimum of twenty seven required by the non-bitfield implementation.

6. Conclusion

In our work developing environment support packages for for the NYU Ultracomputer, we realized the potential benefits of a tool able to atomically read and modify several state variables. This led us to implement fetch-and-add on bitfield integers as presented in this paper. The algorithms which we have developed using this technique have proven to be very efficient. We are presently working on memory

management algorithms which utilize these synchronization tools.

7. Acknowledgments

We greatly appreciate the help of Allan Gottlieb, Jim Lipkis, Isaac Dimitrovsky, Wayne Berke, Brian Anderson, and Lawrence Kaplan for their patience and insights both during the development of the algorithms presented here and of this paper.

References

[Dimitrovsky 85] Isaac Dimitrovsky, *A Short Note on Barrier Synchronization*, Ultracomputer System Software Note #59, Courant Institute, NYU,New York, 1985.

[Dimitrovsky 86] Issac Dimitrovsky, *A Group Lock Algorithm, With Applications*, Ultracomputer Note #112, Courant Institute, NYU,New York, 1986.

[Edler 84] Jan Edler, *Readers/Readers Synchronization*, Ultracomputer System Software Note #46, Courant Institute, NYU,New York, 1984.

Gottlieb 86] Allan Gottlieb, *An Overview of the NYU Ultracomputer Project*, Ultracomputer Note #100, Courant Institute, NYU, New York, 1986.

[GLR 83] Allan Gottlieb, Boris D. Lubachebsky, and Lawrence Rudolph, *Basic Techniques for Efficient Coordination of Very Large Numbers of Cooperating Processors*, ACM TOPLAS 5, pp. 164-189, April 1983b.

[Harrison 86] Malcolm Harrison, *The Add-and-Lambda Operation: An Extension of F&A*, Ultracomputer Note #104, Courant Institute, NYU, New York, 1986. ucn 100

[Harrison 88] Malcolm Harrison *More Uses for Add and Lambda: Eliminating Busy Waits*, Ultracomputer Note #132, Courant Institute, NYU, New York, 1988. Courant Institute, NYU, New York, 1988.

[K&R 78] Brian Kernighan and Dennis M. Ritchie, *The C Programming Language*, (c) 1978 Prentice Hall, Inc., Englewood Cliffs NJ.