Monotonic Counters: A New Mechanism for Thread Synchronization

John Thornley
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
john-t@cs.virginia.edu

K. Mani Chandy
Computer Science Department
California Institute of Technology
Pasadena, California 91125
mani@cs.caltech.edu

Abstract

Only a handful of fundamental mechanisms for synchronizing the access of concurrent threads to shared memory are widely implemented and used. These include locks, condition variables, semaphores, barriers, and monitors. In this paper, we introduce a new synchronization mechanism—monotonic counters—and make a case for its addition to this group. Unlike most other synchronization mechanisms, monotonic counters were designed primarily for multiprocessing, rather than for systems programming. Counters have a very simple definition: a counter object has a nonnegative value, an Increment operation, and a Check operation. Increment atomically increases the counter, and Check suspends until the counter reaches a specified level. We demonstrate that many practical thread synchronization patterns can be expressed more elegantly using counters then with other synchronization mechanisms. Of particular importance, the monotonicity of counters can be used to guarantee deterministic synchronization and the equivalence of multithreaded and sequential execution. In terms of implementation, counters are distinguished from traditional synchronization mechanisms, in that they have a dynamically varying number of thread suspension queues. We give several examples of multithreaded programs that use counter synchronization, and give an implementation of counters on top of locks and condition variables.

1. Introduction

Synchronizing the access of concurrent threads to shared memory in order to prevent unwanted interference is a basic problem in multithreaded programming. Thread synchronization mechanisms provide a means of restricting the possible interleavings of threads, thereby reducing nondeterminacy and allowing stronger program properties to be maintained. For example, locks are often used around update operations on shared data structures to ensure mutual exclusion. The resulting atomicity of the update operations makes it much easier to maintain and reason about the internal consistency of the data structure.

Since Dijkstra first discussed concurrent programming in the 1960s [8], only a small number of fundamentally distinct thread synchronization mechanisms have been widely recognized and implemented. These include locks (also known as mutexes) [23], condition variables (also known as events) [2], semaphores [7], barriers [14], single-assignment variables (also known as sync variables) [4], monitors [13], and rendezvous [1]. Each of these synchronization mechanisms is designed to handle a class of commonly occurring synchronization patterns, with a tradeoff between generality, efficiency, and level of abstraction. No one mechanism alone provides a good solution to all thread synchronization problems.

In this paper, we introduce and describe monotonic counters—a simple yet powerful new thread synchronization mechanism that offers important advantages over existing mechanisms for many practical problems. Unlike most other synchronization mechanisms, monotonic counters (henceforth referred to simply as "counters") were designed primarily for the application of threads to multiprocessing, rather than systems programming. They provide a particularly elegant and efficient mechanism for expressing dataflow style synchronization. For example, one counter can be used in place of an array of condition variables in the synchronization of a writer thread and set of independent reader threads accessing a shared buffer.

The definition of counters is simple. A counter object has three basic attributes: (i) a nonnegative integer value, (ii) an `Increment` operation, and (iii) a `Check` operation. The initial value of the counter is zero. `Increment` atomically increases the value of the counter by a specified amount. `Check` suspends the calling thread until the value of the counter is greater than or equal to a specified level. There is no `Decrement` operation. Since the value of the counter is monotonically increasing, there is no possibility of a race condition occurring on a `Check` operation. This property makes counters well suited to expressing dataflow synchronization.

Many practical synchronization problems can be solved more elegantly (and potentially more efficiently) using counters than with other mechanisms. Counters have a dynamically varying number of thread suspension queues

(corresponding to the different values on which the threads are waiting). This allows a wide variety of sophisticated synchronization patterns to be expressed concisely using only a few counter operations. For example, counter operations can be used as a stronger form of locking, providing sequential ordering in addition to mutual exclusion on a critical section. More generally, `Check` operations can be used to express data dependencies and `Increment` operations can be used to broadcast the availability of data to a set of waiting threads.

Both in their functionality and implementation, counters are distinct from traditional synchronization mechanisms. In terms of functionality, counters are distinguished by their ability to express many-to-many dataflow synchronization using a single synchronization object. Because of their monotonicity, counters can be used in a manner that guarantees deterministic synchronization and the equivalence of sequential and multithreaded execution.

In terms of implementation, counters are distinguished by having a dynamically varying number of thread suspension queues. Locks, condition variables, semaphores, barriers, and single-assignment variables all have a single queue. Practical forms of monitors and rendezvous mechanisms may have many queues, but the number of queues is statically bounded.

2. Counter Programming Interface

In this section, we define the programming interface and describe the operational semantics of monotonic counters. The programming interface for counters, defined as a C++ class, is as follows:

```
class Counter {
public:
   Counter( void ); // Constructor
   ~Counter( void ); // Destructor

   Increment( unsigned int amount );
   // Increments value by amount

   Check( unsigned int level );
   // Suspends until value ≥ level
private:
   unsigned int value;
   ...
};
```

A `Counter` object `c` implicitly has a nonnegative integer attribute `c.value`, which can only be accessed through the public operations on the object. The constructor initializes `c.value` to zero. `c.Check(level)` atomically compares `c.value` to `level` and suspends until `c.value` ≥ `level`. `c.Increment(amount)` atomically increments `c.value` by `amount`, thereby reawakening all `c.Check` operations suspended on values less than or equal to the new `c.value`.

The counter interface operations are chosen to prevent race conditions occurring on counter synchronization. There is no `Decrement` operation. Therefore, the value of a counter is monotonically increasing, and there is no possibility of a `Check` operation missing an `Increment` operation. There is no `Probe` or non-blocking `Check` operation. A choice cannot be based on the instantaneous value of a counter, which may depend on the relative timing of the individual threads. It is impossible for a thread to test the value of a counter without the possibility of suspension.

In a practical programming system, a `Reset` operation might be useful, as a means of efficiently reusing counters between different phases of an algorithm, instead of repeatedly deleting old counters and creating new counters. To avoid the possibility of race conditions, `Reset` must not be called concurrently with other operations on the same counter. `Reset` is not intended as a means of synchronization between threads. Since it is not a fundamental operation, we will discuss counters without `Reset`.

3. Multithreaded Programming Model

In this section, we introduce the multithreaded programming model and notation that we use throughout this paper. Counters can be incorporated in any programming system with explicit thread creation and shared memory, including library-based systems such as Pthreads [17] and Win32 [6] threads, and language-based systems such as OpenMP [19] and Java [18]. We discuss counters in the context of a syntactic variant of Dijkstra's well known `parbegin`–`parend` notation with quantification [8].

The first thread creation construct is the multithreaded block, indicated by the `multithreaded` keyword placed immediately before an ordinary block:

```
multithreaded {
   statement
   ...
   statement
}
```

This notation specifies that the statements of the block should be executed as asynchronous threads. The threads all share the same address space as the parent program. Execution does not continue past the multithreaded block until all the threads have individually terminated. It is illegal for the program to jump between the individual statements of the block, from inside the block to outside the block, or from outside the block to inside the block.

The second thread creation construct is the multithreaded for-loop, indicated by the `multithreaded` keyword placed immediately before an ordinary for-loop:

```
multithreaded
for (int i = expr; i comparison expr; i += expr)
   statement
```

This notation specifies that the iterations of the loop should be executed as asynchronous threads. The threads all share the same address space as the parent program, but each thread has a local copy of the loop control-variable with a different value from the iteration range. In this discussion, the iteration scheme is restricted to a single control-variable and expressions that are not modified within the loop body. Execution does not continue past the multithreaded for-loop until all the threads have individually terminated. It is illegal for the program jump from inside the loop to outside the loop or from outside the loop to inside the loop. In essence, a multithreaded for-loop is a quantified form of multithreaded block.

Multithreaded and ordinary blocks and for-loops can be arbitrarily nested. Further discussion of this model of multithreaded programming can be found in the descriptions of the CC++ [4] and Sthreads [22] notations.

4. Example: All-Pairs Shortest-Paths

In this section, we give a motivating example of the algorithmic and potential performance advantages of counter synchronization. In the example, a counter is used as a less restrictive, and potentially more efficient, replacement for a barrier. The example program is a multithreaded solution to the all-pairs shortest-path problem using the Floyd-Warshall algorithm [11]. Using traditional synchronization mechanisms, this problem can be solved using one barrier or, more efficiently, an array of condition variables. We show how the efficient solution can be implemented using a single counter.

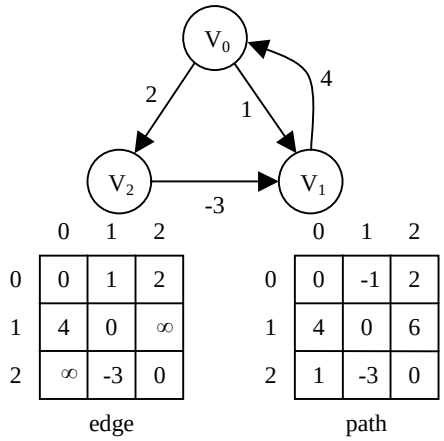

Figure 1. Example of input and output matrices for the all-pairs shortest-path problem.

4.1. The All-Pairs Shortest-Path Problem

The all-pairs shortest-path problem takes as input the edge-weight matrix of a weighted directed graph, and returns the matrix of shortest-length paths between all pairs of vertices in the graph. The graph is required to have no cycles of negative length, and the weight of the edge from a vertex to itself is required to be zero. Figure 1 gives an example of a graph with corresponding input (`edge`) and output (`path`) matrices.

4.2. A Sequential Solution

The following program solves the all-pairs shortest-path problem using the sequential Floyd-Warshall algorithm:

```
void ShortestPaths1(
      int edge[N][N], int path[N][N] )
{
   path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
   for (int k = 0; k < N; k++)
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++) {
            int newPath =
               path[i][k] + path[k][j];
            if (newPath < path[i][j])
               path[i][j] = newPath;
         }
}
```

Initially, `path[i][j]` is assigned `edge[i][j]`, for all `i` and `j`. (For brevity, we use a notational shorthand for array assignment.) After the `k`th iteration, `path[i][j]` is the shortest path from vertex `i` to vertex `j` with intermediate vertices only in vertices `0` to `k`. Therefore, after `N` iterations, `path[i][j]` is the shortest path from vertex `i` to vertex `j` with no restrictions on the intermediate vertices.

4.3. A Multithreaded Solution Using Barriers

The following program solves the all-pairs shortest-path problem using a multithreaded version of the Floyd-Warshall algorithm, with a barrier for synchronization:

```
void ShortestPaths2(
   int edge[N][N], int path[N][N],
   int numThreads )
{
   Barrier b(numThreads);
   path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
   multithreaded
   for (int t = 0; t < numThreads; t++)
      for (int k = 0; k < N; k++) {
         for (int i = t*N/numThreads;
               i < (t+1)*N/numThreads; i++)
            for (int j = 0; j < N; j++) {
               int newPath =
                  path[i][k] + path[k][j];
               if (newPath < path[i][j])
                  path[i][j] = newPath;
            }
         b.Pass();
      }
}
```

The multithreaded outer loop creates `numThreads` threads. Each thread executes the `N` iterations of the Floyd-Warshall algorithm on a subset of the rows of the `path`

matrix. To keep the iterations synchronized, the threads pass through an N-way barrier at the end of each iteration. There are no sharing violations on the concurrent accesses to path, because the algorithm will never assign to path[i][k] or path[k][j] during iteration k.

The barrier algorithm divides the work among an arbitrary number of threads. However, in requiring that all threads complete each iteration before any thread begins the next iteration, the algorithm does not express the full concurrency inherent in the data dependencies. N-way synchronization at the barrier is a bottleneck that creates delays on entry and exit, and processor load imbalance can occur if all threads do not reach the barrier simultaneously.

4.4. A More Efficient Multithreaded Solution Using Condition Variables

The following program solves the all-pairs shortest-path problem using a potentially more efficient multithreaded version of the Floyd-Warshall algorithm, with an array of N condition variables for synchronization:

```
void ShortestPaths3(
    int edge[N][N], int path[N][N],
    int numThreads )
{
   Condition kDone[N];
   int kRow[N][N];

   path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
   kRow[0] = path[0][0..N-1];
   kDone[0].Set();
   multithreaded
   for (int t = 0; t < numThreads; t++)
      for (int k = 0; k < N; k++) {
         kDone[k].Check();
         for (int i = t*N/numThreads;
                  i < (t+1)*N/numThreads; i++) {
            for (int j = 0; j < N; j++) {
               int newPath =
                  path[i][k] + kRow[k][j];
               if (newPath < path[i][j])
                  path[i][j] = newPath;
            }
            if (i == k+1) {
               kRow[k+1][0..N-1] =
                  path[k+1][0..N-1];
               kDone[k+1].Set();
            }
         }
      }
}
```

As with the barrier algorithm, each thread executes the N iterations of the Floyd-Warshall algorithm on a subset of the rows of the path matrix. However, each thread can individually continue with its next iteration as soon as the necessary data is available, instead of waiting for the previous iteration to complete in all the other threads. Condition variable kDone[k] is set when row k of the path matrix has been computed in iteration k-1. Each thread waits on kDone[k] before executing iteration k. To avoid sharing violations, row k of the path matrix computed in iteration k-1 is stored in kRow[k].

The condition variable algorithm avoids the inefficiencies associated with barrier synchronization. Threads synchronize individually, rather than in an N-way bottleneck, and faster threads can execute many iterations ahead of slower threads. Potentially, the N threads can be executing in up to N different iterations. One extra cost of this algorithm is the storage for the kRow matrix. However, the most significant extra cost is allocation of N condition variables. N may be much larger than numThreads.

4.5. The Efficient Solution Using Counters

The following program solves the all-pairs shortest-path problem using the efficient multithreaded version of the Floyd-Warshall algorithm, with a single counter for synchronization, in place of N condition variables:

```
void ShortestPaths3(
   int edge[N][N], int path[N][N],
   int numThreads )
{
   Counter kCount;
   int kRow[N][N];

   path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
   kRow[0] = path[0][0..N-1];
   multithreaded
   for (int t = 0; t < numThreads; t++)
      for (int k = 0; k < N; k++) {
         kCount.Check(k);
         for (int i = t*N/numThreads;
                  i < (t+1)*N/numThreads; i++) {
            for (int j = 0; j < N; j++) {
               int newPath =
                  path[i][k] + kRow[k][j];
               if (newPath < path[i][j])
                  path[i][j] = newPath;
            }
            if (i == k+1) {
               kRow[k+1][0..N-1] =
                  path[k+1][0..N-1];
               kCount.Increment(1);
            }
         }
      }
}
```

Operations on N different values of the single counter replace operations on N different elements of the array of condition variables. The algorithm has the same advantages over the barrier algorithm, without the cost of allocating and maintaining N synchronization objects. Internally, the counter may create synchronization objects for the distinct counter values on which threads are suspended. However, in practice, the number of these objects in existence at any given time is likely to be much less than N.

5. Three Synchronization Patterns

In this section, we describe three examples of practical synchronization patterns that can be expressed more elegantly (and potentially more efficiently) using counters than with traditional synchronization mechanisms. This is far from an complete list of patterns to which counters can usefully be applied. Counters are applicable to many other situations, particularly dataflow style synchronization.

5.1. Ragged Barriers

Counters can often be used to replace traditional barrier synchronization with a less restrictive form of "ragged" barrier. With a ragged barrier, each thread waits at the barrier point only until its own individual data dependencies have been satisfied, instead of until the data dependencies of all threads have been satisfied, as with a traditional barrier. We have already given one example of this pattern in Section 4, with the multithreaded Floyd-Warshall algorithm. In this section, we give a more straightforward example, based on boundary exchange in a time-stepped simulation.

Consider a time-stepped simulation of a one-dimensional object subdivided into `N` cells. The state of internal cell `i` at time `t` is a function of the states of cells `i-1`, `i`, and `i+1` at time `t-1`. The states of the leftmost and rightmost cells remain constant over time. An example is simulation of heat transfer along a metal rod. Similar boundary exchange requirements occur in most multithreaded simulations of physical systems in one or more dimensions. These requirements are traditionally satisfied using barrier synchronization.

The following program implements the simulation using one thread for each cell, with traditional barrier synchronization between threads before cell state exchanges and updates at each time step:

```
float state[N];
Barrier b(N);
...
state[0..N-1] = initial cell states;
multithreaded for(int i = 1; i < N-1; i++) {
   float lState, rState;
   for (int t = 1; t <= numSteps; t++) {
      b.Pass();
      lState  = state[i–1];
      rState = state[i+1];
      b.Pass();
      state[i] = f(lState, state[i], rState);
   }
}
```

All threads synchronize at the barrier twice every time step: once before exchanging cell states, and again before updating cell states. However, complete barrier synchronization between all threads is unnecessarily restrictive. The conditions for safely exchanging and updating the cell states involve dependencies between pairs of neighboring cells, not across all cells. As a consequence of using barriers, the performance of the program is potentially subject to synchronization bottlenecks and load imbalance.

The following program implements the same simulation using an array of counters to provide ragged barrier synchronization between threads:

```
float state[N];
Counter c[N];
...
state[0..N-1] = initial cell states;
c[0].Increment(2*numSteps);
c[N-1].Increment(2*numSteps);
multithreaded for(int i = 1; i < N-1; i++) {
   float myState = state[i], lState, rState
   for (int t = 1; t <= numSteps; t++) {
      c[i-1].Check(2*t-2));
      lState  = state[i–1];
      c[i+1].Check(2*t-2));
      rState = state[i+1];
      c[i].Increment(1);
      myState = f(lState, myState, rState);
      c[i-1].Check(2*t-1);
      c[i+1].Check(2*t-1);
      state[i] = myState;
      c[i].Increment(1);
   }
}
```

As with the traditional barrier algorithm, the threads synchronize every time step before exchanging cell states, and again before updating cell states. However, the synchronization is between pairs of neighboring threads via an array of counters. `c[i].value` $\geq$ `2*t-1` indicates that thread `i` has finished reading both neighboring cell states in time step `t`, and `c[i].value` $\geq$ `2*t` indicates that thread `i` has completed time step `t`. Pairwise synchronization removes the synchronization bottleneck of a traditional barrier and reduces load imbalance by allowing some threads to execute ahead of other threads.

The major cost in the implementation of ragged barriers using counters is the need for `N` counter objects instead of one barrier object. However, the number of counters needed is proportional to the number of threads, not to the problem size. This cost is unlikely to be a practical problem on modern computer systems.

5.2. Mutual Exclusion with Sequential Ordering

Counters can be used as a stronger form of lock synchronization, providing sequential ordering in addition to mutual exclusion on a critical section. With the traditional implementation of mutual exclusion using a pair of lock operations, the order in which threads enter the critical section is nondeterministic. This is desirable in terms of maximizing concurrency, but is undesirable in terms of reasoning, testing, and debugging, and might not satisfy the desired program specification. Replacing the pair of lock operations with a pair of counter operations can guarantee deterministic results.

Consider the computation of a result formed by accumulating a series of independent subresults that are computed concurrently. For example, the result could be a linked list and the `Accumulate` operation could be an append, or the result could be a summation and the `Accumulate` operation could be an addition. Mutual exclusion is required to prevent interference between multiple concurrent `Accumulate` operations on the result.

The following program implements the computation with one thread computing each subresult, and a pair of lock operations to provide mutual exclusion:

```
CompositeItem result;
Lock resultLock;
...
multithreaded for (int i = 0; i < N; i++) {
   SingleItem subresult = Compute(i);
   resultLock.Lock();
   Accumulate(&result, subresult);
   resultLock.Unlock();
}
```

Only one thread can hold `resultLock` at any given time, thereby ensuring mutual exclusion of the `Accumulate` operations. However, if the `Accumulate` operation is not associative and determinacy of results is desired, some other mechanism is required to ensure sequential (or at least deterministic) ordering in addition to mutual exclusion. For example, neither appending an item to a linked list, nor floating point addition are associative operations. With both these examples, the above program may produce different results on repeated executions.

The following program implements the computation with the pair of lock operations replaced with a pair of counter operations, to provide both mutual exclusion and sequential ordering:

```
CompositeItem result;
Counter resultCount;
...
multithreaded for (int i = 0; i < N; i++) {
    SingleItem subresult = Compute(i);
    resultCount.Check(i);
    Accumulate(&result, subresult);
    resultCount.Check(1);
}
```

As with the lock program, only one `Accumulate` operation can execute at any given time. However, the `Accumulate` operations are now additionally constrained to execute in sequential order. `resultCount.value` $\geq$ `i` indicates that thread `i-1` has completed its `Accumulate` operation.

The counter program has greater determinacy at the cost of less concurrency. Counters are a powerful mechanism for providing sequential ordering on top of mutual exclusion in the many cases where determinacy is important and the performance consequences of less concurrency are not great.

5.3. Single-Writer Multiple-Reader Broadcast

Counters can be used to provide elegant, flexible, and efficient dataflow synchronization between a single writer and multiple readers of a sequence of items written to an array. In this synchronization pattern, reading an item does not remove it from the sequence—each reader independently reads the entire sequence. Because a counter has multiple thread suspension queues, a single counter can be used to synchronize the writer thread and any number of independent reader threads, with each thread potentially having a different granularity of synchronization. The writer thread incrementing the counter broadcasts the availability of data to the entire set of reader threads.

The following program demonstrates the single-writer multiple-reader broadcast pattern with synchronization on every item:

```
void Writer(
   Item data[], int n, Counter* dataCount )
{
   for (int i = 0; i < n; i++) {
      data[i] = GenerateItem(i);
      dataCount->Increment(1);
   }
}

void Reader(
   Item data[], int n, Counter* dataCount )
{
   for (int i = 0; i < n; i++) {
      dataCount->Check(i+1);
      UseItem(data[i]);
   }
}

...
Item data[N];
Counter dataCount;

 multithreaded {
   Writer(data, N, &dataCount);
   multithreaded
   for (int r = 0; r < numReaders; r++)
      Reader(data, N, &dataCount);
}
...
```

One `Writer` thread and an arbitrary number of `Reader` threads are executed concurrently, with communication through the shared `data` array, and synchronization through the `dataCount` counter. At any point, some `Reader` threads may be suspended in their `Check` operation, waiting for the `Writer` thread to increment `dataCount`, while other `Reader` threads may be reading `data` items that have previously been written.

Synchronization on every item that is written and read may be too expensive if the time taken to generate and use an item is small. The single-reader multiple-writer broadcast pattern can be generalized to allow the writer and each reader thread to synchronize on a block of items instead of on individual items. The following program adds an indi-

vidual granularity of blocked synchronization to the writer and each reader thread:

```
void Writer(
   Item data[], int n, Counter* dataCount,
   int blockSize )
{
   for (int i = 0; i < n; i++) {
     data[i] = GenerateItem(i);
     if ((i+1)%blockSize == 0)
        dataCount->Increment(blockSize);
   }
   dataCount->Increment(n%blockSize);
}

void Reader(
   Item data[], int n, Counter* dataCount,
   int blockSize )
{
   for (int i = 0; i < n; i++) {
      if (i%blockSize == 0)
         dataCount->Check(min(i+blockSize, n));
      UseItem(data[i]);
   }
}
```

The `Writer` and `Reader` threads now increment and check the `dataCount` counter in multiples of `blockSize` and write and read the `data` array in blocks of items. There is no requirement that `blockSize` be the same in all threads. Different threads can use different blocking granularity based on their individual performance characteristics and requirements. This pattern is now extremely flexible and easily adaptable with regard to practical performance tuning.

The single-writer multiple-reader broadcast pattern is a dataflow synchronization pattern that occurs in many diverse applications of threads to multiprocessing. For example, in the Paraffins Problem [9], an array of molecules of a certain size can be generated by one thread and concurrently read by other threads that in turn generate arrays of larger molecules. The pattern is different from the multiple-writers multiple-readers bounded-buffer problem [16], which is elegantly solved using semaphores. Just as counters are not well suited to implementing bounded buffers, semaphores and other traditional synchronization mechanisms are not well suited to implementing the single-writer multiple-reader broadcast pattern.

6. Monotonicity, Determinacy, and Sequential Equivalence

In this section, we discuss the monotonicity of counters, and briefly outline how this property helps us guarantee deterministic synchronization and the equivalence of multithreaded and sequential execution. If shared variables are guarded against concurrent operations, a program that uses only counter synchronization is guaranteed to produce deterministic results on all executions. Moreover, if sequential execution of the program (i.e., execution ignoring the `multithreaded` keyword) does not deadlock, multithreaded execution is guaranteed not to deadlock and to produce the same results as sequential execution. These properties are extremely useful in the testing and debugging of multithreaded programs.

Even when shared variables are guarded against concurrent operations, traditional synchronization mechanisms can introduce nondeterminacy into a program through timing dependent race conditions between threads. For example, consider the following program that uses a lock:

```
multithreaded {
   {xLock.Lock(); x = x+1; xLock.Unlock();}
   {xLock.Lock(); x = x*2; xLock.Unlock();}
}
```

Even though there are no concurrent operations on `x`, the resulting value of `x` is nondeterministic because of the race condition on the order in which the two threads acquire the lock. In contrast, because counters are monotonic, once a synchronization condition is enabled it remains enabled, and there is no possibility of a race condition to catch or miss a particular counter value. For example, consider the following program that uses a counter:

```
multithreaded {
   {xCount.Check(0); x = x+1; xCount.Increment(1);}
   {xCount.Check(1); x = x*2; xCount.Increment(1);}
}
```

The resulting value of `x` is deterministic, because the `Check` operations will succeed in the same order in all executions and the operations on `x` will occur in the same order. Moreover, since sequential execution does not deadlock, multithreaded execution cannot deadlock and will always be equivalent to sequential execution.

Programs that use only counter synchronization can still be erroneously nondeterministic if they do not guard against concurrent access to shared variables. For example, consider the following program using a counter:

```
multithreaded {
   {xCount.Check(0); x = x+1; xCount.Increment(1);}
   {xCount.Check(0); x = x*2; xCount.Increment(1);}
}
```

The result of the program is nondeterministic because of the possibility of concurrent execution of operations on `x`. The nondeterminacy is caused by concurrent access to a shared variable, not by a synchronization race condition.

Although beyond the scope of this paper, the conditions to prevent concurrent access to shared variables using counters are straightforward [21]. Essentially, each pair of operations on a shared variable must be separated by a transitive chain of counter operations. If these conditions can be shown to hold in any one execution of the program, they must hold in all executions of the program. Therefore, if sequential execution satisfies the conditions, multithreaded execution is also guaranteed to satisfy the condi-

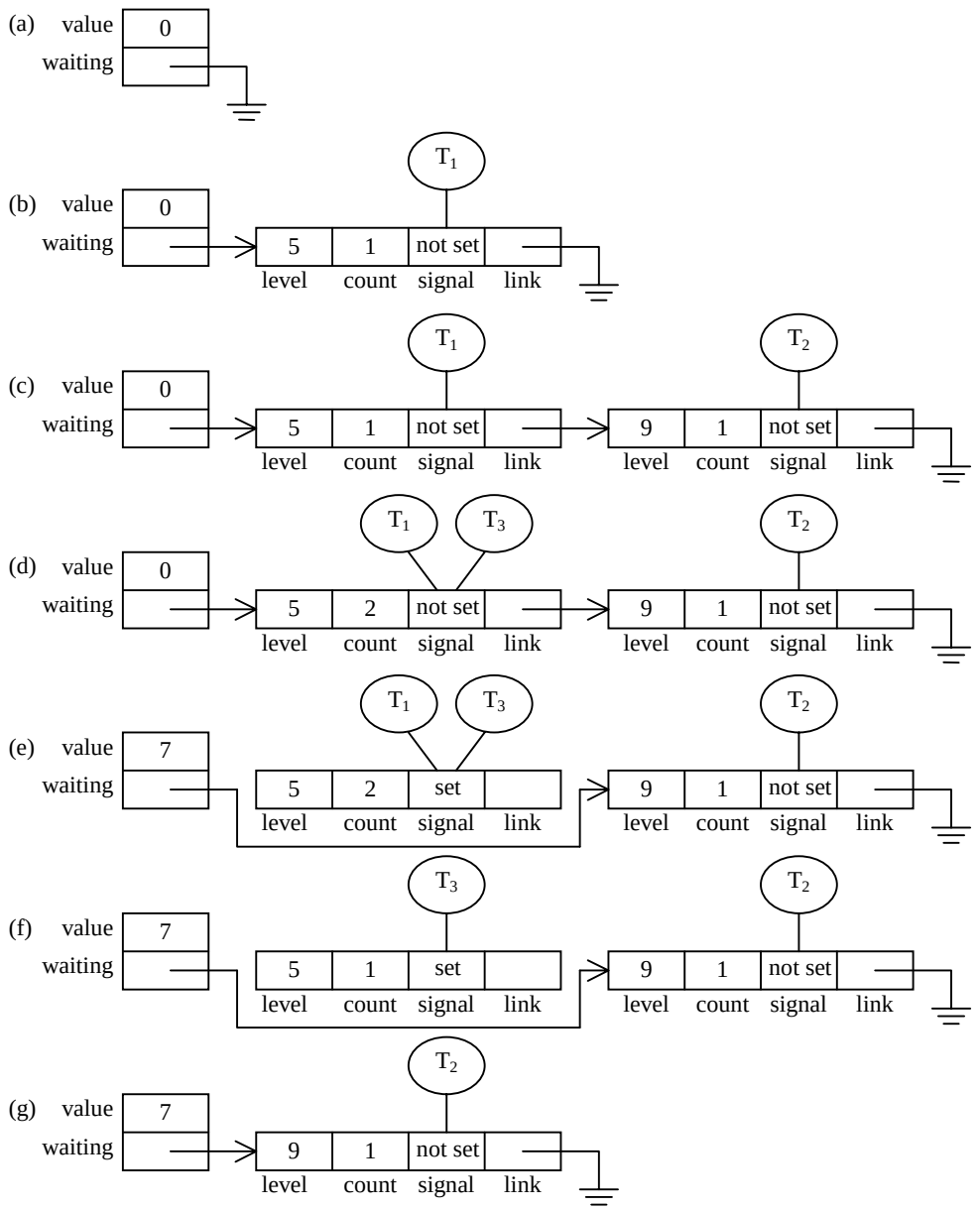

Figure 2. The structure of counter `c` after: (a) construction, (b) `c.Check(5)` by thread T_1, (c) `c.Check(9)` by thread T_2, (d) `c.Check(5)` by thread T_3, (e) `c.Increment(7)` by T_0, (f) T_1 resumes execution, (g) T_3 resumes execution.

tions, and hence produce the same results as sequential execution. This result forms the basis of a powerful methodology for developing multithreaded programs using sequential reasoning, testing, and debugging [22].

All the programs using counters that we have presented in this paper satisfy the conditions on shared variables, therefore are guaranteed to be deterministic. In addition, the programs for mutual exclusion with sequential ordering in section 5.2 and single-reader single-writer broadcast in section 5.3 have equivalent multithreaded and sequential execution. The cost of increased determinacy is decreased concurrency. Synchronization using counters provides an effective means of controlling the tradeoff between determinacy and concurrency.

7. Implementation and Efficiency

In this section, we outline an efficient implementation of counters on top of traditional locks and condition variables. The key data structure is a dynamically changing ordered list of condition variables, with one node for each level on which one or more threads are waiting. A lock provides the necessary mutual exclusion within operations on the list. The storage for a counter and the time com-

plexity of operations on a counter are proportional to the number of different levels on which threads are waiting, not to the total number of waiting threads.

A counter is represented as a structure with three components: (i) the nonnegative integer counter value, (ii) an ordered linked list of dynamically allocated nodes representing the counter levels on which threads are waiting, and (iii) a lock for mutual exclusion during update operations. Initially, the counter value is zero, the waiting list is null, and the lock is not held. Each node in the waiting list is a structure with four components: (i) a level on which threads are waiting, (ii) a count of the number of threads waiting at that level, (iii) a condition variable on which the threads wait, and (iv) a link to the next node in the list. The waiting list is ordered on the levels, and never contains levels less than or equal to the counter value.

`Check` with a level less than or equal to the current counter value returns immediately. `Check` with a level greater than the counter value searches the waiting list for a node at the specified level, and inserts a new node if none exists at that level. (In a new node, the count is initially zero and the condition variable is not set.) Then the count of the node is then increased by one and a wait operation is performed on the condition variable, thereby suspending the calling thread.

`Increment` increases the counter value by the specified amount, then removes all nodes with levels less than or equal to the new counter value from the waiting list. The condition variable is set in each of these nodes, which wakes up all threads waiting at those levels. When each woken thread continues execution (in the `Check` operation), it decrements the count in the node. The thread that decrements the count to zero deallocates the node. Figure 2 gives an example of the structure of a counter over a series of `Check` and `Increment` operations.

The storage requirements of a counter are proportional to the number of different levels at which threads are waiting. In particular, the number of condition variables is proportional to this number. The time complexity of `Check` and `Increment` operations is also proportional to the number of different levels at which threads are waiting, not to the total number of waiting threads. Although the number of different levels on which threads wait over the lifetime of the counter may be high, the number of levels at which threads are waiting at any given time is likely to be much lower.

8. Comparison with Related Work

In this section, we compare counters with other synchronization mechanisms, and particularly with other similarly motivated multithreaded programming models.

Counters are not intended as a replacement for traditional synchronization mechanisms, e.g., we do not propose that locks and condition variables be replaced by counters. Most traditional synchronization mechanisms were designed for systems programming, where nondeterminacy is an inherent feature of the problem space. Counters were designed primarily for multiprocessing, where deterministic synchronization offers many benefits in program development. We propose that counters be recognized as a fundamental synchronization mechanism for this application domain, and be provided alongside other synchronization mechanisms in general-purpose multithreaded programming systems.

The idea of monotonic and deterministic synchronization mechanisms derives from declarative and dataflow programming. Parallel dataflow languages such as Val [15] and Sisal [10], and parallel logic programming languages such as Concurrent Prolog [20], Parlog [5], and Strand [12] are based on single-assignment variables. PCN [3] and CC++ incorporate both mutable and single-assignment variables, and both deterministic and nondeterministic synchronization within an integrated framework. Counters extend this model by (i) separating the synchronization and data-holding functionality, and (ii) allowing synchronization on many different values of a single object. In addition, counters are not tied to any particular notation or type system—they can easily be incorporated in almost any language as a library.

Counters are distinguished from other synchronization mechanisms in that they have a dynamically varying number of thread suspension queues. This allows sophisticated synchronization patterns to be elegantly expressed using only a few counter objects. Other synchronization mechanisms typically have either one thread suspension queue (e.g., locks, condition variables, semaphores, and single-assignment variables) or a statically bounded number of queues (e.g., monitors and rendezvous).

9. Conclusion

We have introduced and demonstrated monotonic counters—a simple yet powerful new mechanism for dataflow style thread synchronization. Whereas most widely used synchronization mechanisms were designed for systems programming, counters were designed primarily for multiprocessing applications of threads. Many practical problems from this application domain can be solved more elegantly and potentially more efficiently using counters than with traditional synchronization mechanisms. For example, counters can be used to implement a less restrictive form of barrier, and a stronger form of mutual exclusion with sequential ordering.

Because the value of a counter is monotonically increasing, there can be no race conditions to catch a value, and synchronization is therefore deterministic. Determinacy greatly simplifies testing, debugging, and reasoning about correctness. Moreover, in many cases, multithreaded

execution of a program that uses counters can be guaranteed to be equivalent to sequential execution. In these cases, multithreaded programs can be developed using standard sequential methods and tools.

Much of the expressive power of counters is a result of the property that each counter object has a dynamically varying number of thread suspension queues. Traditional synchronization mechanisms have either a single queue or a statically bounded number of queues. Therefore, one counter can often replace many traditional synchronization objects, and one counter operation often corresponds to many traditional synchronization operations.

Counters can be efficiently implemented on top of traditional synchronization mechanisms such as locks and condition variables. The storage requirements and time complexity are proportional to the number of different values on which threads are suspended at any given time, not to the total number of suspended threads. In terms of both implementation and programming model, counters can easily be integrated with traditional synchronization mechanisms within a single framework.

The combination of determinacy and multiple thread synchronization queues makes counters a very elegant, practical, and powerful new thread synchronization mechanism. We believe that monotonic counters should be considered for widespread inclusion in general-purpose thread libraries and multithreaded languages.

Acknowledgements

The counter solution to the all-pairs shortest-path problem was first suggested by Hiroshi Ishii. Thanks to Microsoft Research University Programs, through Dave Ladd, for supporting this work. Thanks also to Margaret Thornley for proofreading this paper. This research was funded in part by the NSF CRPC under Cooperative Agreement No. CCR-9120008 and by the Army Research Laboratory under Contract No. DAHC94-96-C-0010.

References

[1] American National Standards Institute. Inc. *The Programming Language Ada Reference Manual*. Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A.

[2] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[3] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett. Boston, Massachusetts, 1993.

[4] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 281–313. MIT Press, Cambridge, Massachusetts, 1993.

[5] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, volume 8, number 1, pages 1–49, January 1986.

[6] Aaron Cohen and Mike Woodring. *Win32 Multithreaded Programming*. O'Reilly, Sebastapol, California, 1998.

[7] E. W. Dijkstra. The structure of the "THE" multiprogramming system. *Communications of the ACM*, volume 11, number 5, pages 341–346, May 1968.

[8] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, New York, 1968.

[9] J. T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. North-Holland, Amsterdam, The Netherlands, 1992.

[10] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, volume 10, number 4, pages 349–366, December 1990.

[11] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, volume 5, number 6, page 345, June 1962.

[12] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[13] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, volume 17, number 10, pages 549–557, October 1974.

[14] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, volume 14, pages 125–169, 1984.

[15] James R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, volume 4, number 1, pages 44–82, January 1982.

[16] E. Morenoff and J. B. McLean. Inter-program communication, program string structures and buffer files. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 175–183, 1967.

[17] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, Sebastapol, California, 1996.

[18] Scott Oaks and Henry Wong. *Java Threads, second edition*. O'Reilly, Sebastapol, California, 1998.

[19] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*. http://www.openmp.org/, October 1998.

[20] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, volume 19, number 8, pages 44–58, August 1986.

[21] John Thornley. *A Parallel Programming Model with Sequential Semantics*. Ph.D. thesis, California Institute of Technology, May 1996. Caltech CS-TR-96-12.

[22] John Thornley, K. Mani Chandy, and Hiroshi Ishii. A system for structured high-performance multithreaded programming in Windows NT. In *2nd USENIX Windows NT Symposium Proceedings*, pages 67–76, Seattle, Washington, August 3–5, 1998.

[23] N. Wirth. On multiprogramming, machine coding, and computer organization. *Communications of the ACM*, volume 12, number 9, pages 489–98, September 1969.