0.1 THE SHARED DATA PATTERN

Problem

How does one explicitly manage shared data accessed with total operations\(^1\) inside a set of concurrent tasks?

Context

Most of the Concurrent Algorithm Strategy patterns simplify the handling of shared data by using techniques to “pull” the shared data “outside” the set of tasks. Examples include replication plus reduction in the Task Parallelism pattern and alternating computation and communication in the Geometric Decomposition pattern. For certain problems, however, these techniques do not apply, thereby requiring that shared data be explicitly managed inside the set of concurrent tasks.

For example, consider the phylogeny problem from molecular biology, as described in [YWC+96]. A phylogeny is a tree showing relationships between organisms. The problem consists of generating large numbers of subtrees as potential solutions and then rejecting those that fail to meet the various consistency criteria. Different sets of subtrees can be examined concurrently, so a natural task definition in a parallel phylogeny algorithm would be the processing required for each set of subtrees. However, not all sets must be examined—if a set \(S\) is rejected, all supersets of \(S\) can also be rejected. Thus, it makes sense to keep track of the sets still to be examined and the sets that have been rejected. Given that the problem naturally decomposes into nearly independent tasks (one per set), the solution to this problem would use the Task Parallelism pattern. Using the pattern is complicated, however, by the fact that all tasks need both read and write access to the data structure of rejected sets. Also, because this data structure changes during the computation, we cannot use the replication technique described in the Task Parallelism pattern. Partitioning the data structure and basing a solution on this data decomposition, as described in the Geometric Decomposition pattern, might seem like a good alternative, but the way in which the elements are rejected is unpredictable, so any data decomposition is likely to lead to a poor load balance.

Similar difficulties can arise any time shared data must be explicitly managed inside a set of concurrent tasks. The common elements for problems that need the Shared Data pattern are (1) at least one data structure is accessed by multiple tasks in the course of the program’s execution, (2) at least one task modifies the shared data structure, and (3) the tasks potentially need to use the modified value during the concurrent computation.

\(^1\)Total operations have a well-defined return value in any state. In contrast partial operations may cause blocking until a returned values can be determined. For example, the take operation might be chosen to be total if, when invoked on an empty queue it returns immediately with an indication that the queue was empty, or partial if it blocks until an item is added to the queue.
Forces

- The results of the computation must be correct for any ordering of accesses to the shared data that could occur during the computation.
- Explicitly managing shared data can incur parallel overhead, which must be kept small if the program is to run efficiently.
- Techniques for managing shared data can limit the number of tasks that can run concurrently, thereby reducing the potential scalability of an algorithm.
- If the constructs used to manage shared data are not easy to understand, the program will be more difficult to maintain.

Solution

Explicitly managing shared data can be one of the more error-prone aspects of designing a parallel algorithm. Therefore, a good approach is to start with a solution that emphasizes simplicity and clarity of abstraction and then try more complex solutions if necessary to obtain acceptable performance. The solution reflects this approach.

*Be sure this pattern is needed*

The first step is to confirm that this pattern is truly needed; it might be worthwhile to revisit decisions made earlier in the design process (the decomposition into tasks, for example) to see whether different decisions might lead to a solution that fits one of the Concurrent Algorithm Strategy patterns without the need to explicitly manage shared data. For example, if the Task Parallelism pattern is a good fit, it is worthwhile to review the design and see if dependencies can be managed by replication and reduction.

*Define an abstract data type*

Assuming this pattern must indeed be used, start by viewing the shared data as an abstract data type (ADT) with a fixed set of (possibly complex) operations on the data. For example, if the shared data structure is a queue (see the Shared Queue pattern), these operations would consist of put (enqueue), take (dequeue), and possibly other operations, such as a test for an empty queue or a test to see if a specified element is present. Each task will typically perform a sequence of these operations. These operations should have the property that if they are executed serially (that is, one at a time, without interference from other tasks), each operation will leave the data in a consistent state.

The implementation of the individual operations will most likely involve a sequence of lower-level actions, the results of which should not be visible to other UEs. For example, if we implemented the previously mentioned queue using a linked list, a take operation actually involves a sequence of lower-level operations (which may themselves consist of a sequence of even lower-level operations):
1. Use variable `first` to obtain a reference to the first object in the list.
2. From the first object, get a reference to the second object in the list.
3. Replace the value of `first` with the reference to the second object.
4. Update the size of the list.
5. Return the first element.

If two tasks are executing “take” operations concurrently, and these lower-level operations are interleaved (that is, the “take” operations are not being executed atomically), the result could easily be an inconsistent list.

**Implement an appropriate concurrency-control protocol**

After the ADT and its operations have been identified, the objective is to implement a concurrency-control protocol to ensure that these operations give the same results as if they were executed serially. There are several ways to do this; start with the first technique, which is the simplest, and then try the other more complex techniques if it does not yield acceptable performance. These more complex techniques can be combined if more than one is applicable.

**One-at-a-time execution.**

The easiest solution is to ensure that the operations are indeed executed serially.

In a shared-memory environment, the most straightforward way to do this is to use a mutual-exclusion protocol to ensure that only one UE at a time is accessing the shared data. Exactly how this is implemented will depend on the facilities of the target programming environment. Typical choices include mutex locks, synchronized blocks, critical sections, and semaphores. These mechanisms are described in *Concurrency Foundation Constructs*. If the programming language naturally supports the implementation of abstract data types, it is usually appropriate to implement each operation as a procedure or method, with the mutual-exclusion protocol implemented in the methods.

Another way to ensure one-at-a-time execution is to assign the shared data structure to a particular UE. Other UEs that need to access the shared data issue a request to the designated UE, which then processes the requests serially. This is a natural approach in message passing systems. It is also a common way of structuring GUI toolkits. For example, in both Java Swing [Swi] and SWT [SWT] the GUI data structures are not thread safe; instead of accessing these directly, applications post requests to update the GUI data structures to an event queue. The event queue also receives requests corresponding to GUI events such as mouse clicks, key presses, and timer expirations. These requests are processed serially by the event dispatch thread.

In either environment, this approach is usually not difficult to implement, but it can be overly conservative (that is, it might disallow concurrent execution of operations that would be safe to execute simultaneously), and it can produce a bottleneck that negatively affects the performance of the program. If this is the case, the remaining approaches described in this section should be reviewed to
see whether one of them can reduce or eliminate this bottleneck and give better performance.

**Noninterfering sets of operations.**

First, analyze the potential *interference* between the operations. Two operations performed by different UEs interfere if one of them modifies a shared variable that is read or modified by the other. Note that an operation may interfere with itself if it can be executed concurrently by multiple UEs. If operations cannot interfere, then they do not need to exclude each other.

Sometimes the operations fall into disjoint sets, where the operations in different sets do not interfere with each other. In this case, the amount of concurrency can be increased by treating each of the sets as a different critical section. That is, within each set, operations execute one at a time, but operations in different sets can proceed concurrently.

**Readers/writers.**

If there is no obvious way to partition the operations into disjoint non-interfering sets, consider the type of interference. It may be the case that some of the operations modify the data, but others only read it. For example, if operation A is a writer (both reading and writing the data) and operation B is a reader (reading, but not writing, the data), A interferes with itself and with B, but B does not interfere with itself. Thus, if one task is performing operation A, no other task should be able to execute either A or B, but any number of tasks should be able to execute B concurrently. In such cases, it may be worthwhile to implement a readers/writers protocol that will allow this potential concurrency to be exploited. The overhead of managing the readers/writers protocol is greater than that of simple mutex locks, so the length of the readers’ computation should be long enough to make this overhead worthwhile. In addition, reading should typically occur much more frequently than writing.

The *java.util.concurrent* package provides read/write locks to support the readers/writers protocol. The code in Fig. 1 illustrates how these locks are typically used: First instantiate a *ReadWriteLock*, and then obtain its read and write locks. *ReentrantReadWriteLock* is a class that implements the *ReadWriteLock* interface. To perform a read operation, the read lock must be locked. To perform a write operation, the write lock must be locked. The semantics of the locks are that any number of UEs can simultaneously hold the read lock, but the write lock is exclusive; that is, only one UE can hold the write lock, and if the write lock is held, no UEs can hold the read lock either.

Readers/writers protocols are discussed in [And00, GPB+06] and most operating systems texts.

**Reducing the size of the critical section.**

Another approach to improving performance begins with analyzing the implementations of the operations in more detail. It may be the case that only part of the operation involves actions that interfere with other operations. If so, the size of
class X {
    ReadWriteLock rw = new ReentrantReadWriteLock();
    // ...

    /*operation A is a writer*/
    public void A() throws InterruptedException {
        rw.writeLock().lock(); // lock the write lock
        try {
            // ... do operation A
        }
        finally {
            rw.writeLock().unlock(); // unlock the write lock
        }
    }

    /*operation B is a reader*/
    public void B() throws InterruptedException {
        rw.readLock().lock(); // lock the read lock
        try {
            // ... do operation B
        }
        finally {
            rw.readLock().unlock(); // unlock the read lock
        }
    }
}

Figure 1: Typical use of read/write locks. These locks are defined in the java.util.concurrent.locks package. Putting the unlock in the finally block ensures that the lock will be unlocked regardless of how the try block is exited (normally or with an exception) and is a standard idiom in Java programs that use locks rather than synchronized blocks.
the critical section can be reduced to that smaller part. Notice that this sort of optimization is very easy to get wrong, so it should be attempted only if it will give significant performance improvements over simpler approaches, and the programmer completely understands the interferences in question.

**Nested locks.**

This technique is a hybrid between two of the previous approaches, noninterfering operations and reducing the size of the critical section. Suppose we have an ADT with two operations. Operation A does a lot of work both reading and updating variable x and then reads and updates variable y in a single statement. Operation B reads and writes y. Some analysis shows that UEs executing A need to exclude each other, UEs executing B need to exclude each other, and because both operations read and update y, technically, A and B need to mutually exclude each other as well. However, closer inspection shows that the two operations are almost noninterfering. If it were not for that single statement where A reads and updates y, the two operations could be implemented in separate critical sections that would allow one A and one B to execute concurrently. A solution is to use two locks, as shown in Fig. 2. A acquires and holds lockA for the entire operation. B acquires and holds lockB for the entire operation. A acquires lockB and holds it only for the statement updating y.

```java
class Y {
    Object lockA = new Object();
    Object lockB = new Object();

    void A() {
        synchronized(lockA) {
            ....compute....
            synchronized(lockB) {
                ....read and update y....
            }
        }
    }

    void B() throws InterruptedException {
        synchronized(lockB) {
            ....compute....
        }
    }
}
```

Figure 2: Example of nested locking using synchronized blocks with dummy objects lockA and lockB

Whenever nested locking is used, the programmer should be aware of the potential for deadlocks and double-check the code. (The classic example of deadlock, stated in terms of the previous example, is as follows: A acquires lockA and B
acquires lockB. A then tries to acquire lockB and B tries to acquire lockA. Neither operation can now proceed.) Deadlocks can be avoided by assigning a partial order to the locks and ensuring that locks are always acquired in an order that respects the partial order. In the previous example, we would define the order to be lockA < lockB and ensure that lockA is never acquired by a UE already holding lockB.

Application-specific semantic relaxation.

Yet another approach is to consider partially replicating shared data (the software caching described in [YWC+96]) and perhaps even allowing the copies to be inconsistent if this can be done without affecting the results of the computation. For example, a distributed-memory solution to the phylogeny problem described earlier might give each UE its own copy of the set of sets already rejected and allow these copies to be out of synch; tasks may do extra work (in rejecting a set that has already been rejected by a task assigned to a different UE), but this extra work will not affect the result of the computation, and it may be more efficient overall than the communication cost of keeping all copies in synch.

Speculative (optimistic) updates

The lock-based solution described so far ensures that interference between UEs will never cause inconsistent data by serializing access to the data using critical sections or locks. This often results in code with the following basic structure:

```
acquire lock
read shared variables
compute new values
update shared variables
release lock
```

An alternative approach replaces holding a lock during the entire read-compute-update operation with a requirement to atomically read the shared data, and atomically check if the shared data has been modified since reading and update as shown below:

```
acquire lock
read shared variables
lock modified
compute new values
update shared variables
release lock
```
success = false;
do
  atomically read shared variables
  compute new values, stores in local variable
  atomically
    check if the data has changed since it was read
    update shared variables
  success = true;
while (! success)

This approach is called speculative, or optimistic because, rather than preventing interference by completely serializing the read-compute-update steps, interference is allowed to happen. If interference does happen, it must be detected, and if necessary, the entire operation retried.

One way of implementing this is to use locks to implement the atomic operations. The potential advantage over previous approaches is that the locks are only held long enough to atomically read the data, and then later to check and write the updates, thus allowing more concurrency with the tradeoff that when contention does occur, the penalty can be significant. It is only appropriate if checking that data has not been changed since it was read can be done efficiently and conflicts are infrequent. Also, livelock, where the UEs get in a situation where they repeatedly interfere and retry with none making forward progress, is at least a theoretical possibility. Delaying a thread a random amount of time before trying again prevents in practice. The most common approach is an exponential backoff scheme where the mean delay becomes longer after each retry.

The Transactional Memory pattern, offers a convenient and robust way to implement speculative updates. In transactional memory systems, some combination of the run-time system and hardware \(^2\), automates detection of conflicting operations and rolling back when conflicts occur. The programmer, to a first approximation (but see the Transactional Memory pattern for caveats), only needs to indicate the block of code that should be executed atomically along the lines of

\[
\text{atomic}(
  \text{read, compute, and update shared variables}
)
\]

The programmer still needs to take care to make the atomic section as small as possible, but most of the other considerations, including deadlocks, are no relevant.

\(^2\)The hardware used for detecting conflicts is similar to that used to implement cache coherency protocols.
Non-blocking single variable updates

Non-blocking protocols have the property that the delay of one UE cannot delay the others. There are several non-blocking variations on the speculative approach that can be applied when appropriate mechanisms are available. (Clearly, non-blocking protocols must be implemented without using locks.) A useful situation in practice is when the shared data is a single variable and an atomic read-modify-write instruction is available. It is not uncommon to use low level bit manipulation to combine distinct values into a single variable so that this solution can be used. Modern architectures typically provide a hardware compare-and-swap (CAS) or load-locked/store-conditional (LL/SC) instructions which may or may not be conveniently accessible to the programmer in a particular parallel programming environment. For example, Java offers CAS on boolean, int, long, and reference types in the form of the `compareAndSet` method found in the `java.util.concurrent.atomic` package and C# provides it by the `Interlocked.CompareExchange` method.

To illustrate how such a feature can be used, suppose that the variable to be updated is a long value that is represented using an `AtomicLong` object. Then `shared` can be updated with the `compareAndSet` method where the method `boolean compareAndSet(long expect, long update)` compares `expect` to the current value of the variable and if they are the same, updates the variable with `newVal` and returns `true`. If the current value is not equal to `expect`, then `compareAndSet` leaves its value unchanged and returns false. All of this is done in a single atomic step.

The speculative approach can be reformulated as

```java
AtomicLong shared = ... //the shared variable
...
do {
    long oldVal = shared.get();
    long newVal = compute(oldVal);
} while (! shared.compareAndSet(oldVal, newVal));
```

The CAS operation does not succeed if the value of the shared variable has changed since it was read. This algorithm is non-blocking, since there is no way that one UE will be blocked waiting for another, possibly suspended, UE to take an action (such as releasing a lock). In addition, it is the case that even in the presence of contention, some UE will make progress since the only way for one UE’s CAS to fail is if another’s has succeeded.

Non-blocking updates with immutable objects

If the shared data does not fit into a single variable, but can be packaged in a record or object that is treated as a unit and referenced via a reference or pointer, then the non-blocking approach described in the previous subsection may still be useful.
The key requirement is that the fields of the record or object are never changed once they become visible to other UEs. Instead, a new record or object is created. This has the effect that the mutable part of the shared data is a single variable, the pointer or reference, which can be updated as before with CAS. Since the fields are immutable, object identity rather than the actual content of the fields can be checked.

```java
class Complex{
    final int re;
    final int imag;
    public Complex(int re, int imag){this.re = re; this.imag = imag;}
}

class AppUsingComplex{
    AtomicReference sharedComplex = ....;
    do {
        Complex oldVal = sharedComplex.get();
        int re = oldVal.re;
        int im = oldVal.imag;
        ...compute new values using re and im...
        Complex newVal = new Complex (newRe, newImag);
        while (! sharedComplex.compareAndSet(oldVal,newVal);)
    }
}
```

Figure 3: An immutable class for complex numbers illustrates non-blocking updates with immutable objects.

Of course, it isn’t always appropriate to make a new copy of a shared data structure every time it changes.

**General non-blocking data structures**

For a few important data structures, where the above approaches are not appropriate, practical non-blocking algorithms have been devised. These update the data structure using a sequence of non-blocking atomic operations such as read, write, and CAS, that operate on a single variable. These algorithms are typically fairly intricate because they must be correct regardless of how the atomic actions from concurrent updates by different UEs are interleaved. In addition, they must be devised so that delaying one UE (due to thread scheduling or even failure) cannot prevent other UEs from making progress. The approach usually taken is to set up things so that any UE can complete any other UE’s partially completed update. Several such implementations be found in the java.util.concurrent package and examples are given in the Shared Queue and Shared Map patterns. Due to their

---

3It is possible to systematically generate a non-blocking algorithm for any data structure [Her91,Her92] but algorithms obtained this way are not efficient enough to be practical in most cases.
complexity, design of general non-blocking algorithms is usually best left for experts and concurrent programming library developers.

**Review other considerations**

**Memory synchronization.**

Make sure memory is synchronized as required: Caching and compiler optimizations can result in unexpected behavior with respect to shared variables. For example, a stale value of a variable might be read from a cache or register instead of the newest value written by another task, or the latest value might not have been flushed to memory and thus would not be visible to other tasks. In most cases, memory synchronization is performed implicitly by higher-level synchronization primitives, but it is still necessary to be aware of the issue. Unfortunately, memory synchronization techniques are very platform-specific. In OpenMP, the flush directive can be used to synchronize memory explicitly; it is implicitly invoked by several other directives. In Java, memory is implicitly synchronized when entering and leaving a synchronized block and when locking and unlocking locks. Also, variables marked `volatile` are implicitly synchronized with respect to memory. Operations such as `get`, `set`, and `compareAndSet` in the classes in the `java.util.concurrent.atomic` package also have memory synchronization effects equivalent to reading, writing, or reading and writing a volatile. This is discussed in more detail in the *Concurrency Foundation Constructs* design space.

**Task scheduling.**

Consider whether the explicitly managed data dependencies addressed by this pattern affect task scheduling. A key goal in deciding how to schedule tasks is good load balance; in addition to the considerations described in the *Concurrent Algorithm Strategy* pattern being used, one should also take into account that tasks might be suspended waiting for access to shared data. It makes sense to try to assign tasks in a way that minimizes such waiting, or to assign multiple tasks to each UE in the hope that there will always be one task per UE that is not waiting for access to shared data.

**Examples**

**Shared collections**

The Java Collections Framework found in the `java.util` package and `java.util.concurrent` packages provide numerous examples of the *Shared Data* pattern. The classes in the `java.util` package provide several basic non-thread-safe implementations of sets, lists, and maps. The easiest way to obtain a thread-safe version of the ADT is to use a wrapper class that ensures mutually exclusive access to each of the methods. The `java.util.Collections` class provides factory methods for these thread-safe classes. The `java.util.concurrent` package provides more elaborate, better performing implementations of these ADTs.
Genetic algorithm for nonlinear optimization

Consider the GAFORT program from the SPEC OMP2001 benchmark suite [ADE+01]. GAFORT is a small Fortran program (around 1,500 lines) that implements a genetic algorithm for nonlinear optimization. The calculations are predominantly integer arithmetic, and the program’s performance is dominated by the cost of moving large arrays of data through the memory subsystem.

The details of the genetic algorithm are not important for this discussion. We are going to focus on a single loop within GAFORT. Pseudo code for the sequential version of this loop, based on the discussion of GAFORT in [EM01], is shown in Fig. 4. This loop shuffles the population of chromosomes and consumes on the order of 36 percent of the runtime in a typical GAFORT job [AE03].

```
Int const NPOP // number of chromosomes (~40000)
Int const NCHROME // length of each chromosome

Real :: tempScalar
Array of Real :: temp(NCHROME)
Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: fitness(NPOP)
Int :: j, iother

loop [j] over NPOP
    iother = rand(j) // returns random value greater
                  // than or equal to zero but not
                  // equal to j and less than NPOP
    // Swap Chromosomes
    temp(1:NCHROME) = iparent(1:NCHROME, iother)
    iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
    iparent(1:NCHROME, j) = temp(1:NCHROME)
    // Swap fitness metrics
    tempScalar = fitness(iother)
    fitness(iother) = fitness(j)
    fitness(j) = tempScalar
end loop [j]
```

Figure 4: Pseudocode for the population shuffle loop from the genetic algorithm program GAFORT

A parallel version of this program will be created by parallelizing the loop, using the Loop Parallelism pattern. In this example, the shared data consists of the iparent and fitness arrays. Within the body of the loop, calculations involving these arrays consist of swapping two elements of iparent and then swapping the corresponding elements of fitness. Examination of these operations shows that two swap operations interfere when at least one of the locations being swapped is the same in both operations.

Thinking about the shared data as an ADT helps us to identify and analyze
the actions taken on the shared data. This does not mean, however, that the implementation itself always needs to reflect this structure. In some cases, especially when the data structure is simple and the programming language does not support ADTs well, it can be more effective to forgo the encapsulation implied in an ADT and work with the data directly. This example illustrates this.

As mentioned earlier, the chromosomes being swapped might interfere with each other; thus the loop over \( j \) cannot safely execute in parallel. The most straightforward approach is to enforce a “one at a time” protocol using a critical section, as shown in Fig. 5. It is also necessary to modify the random number generator so it produces a consistent set of pseudorandom numbers when called in parallel by many threads. The algorithms to accomplish this are well understood [Mas97], but will not be discussed here.

The program in Fig. 5 can safely execute with multiple threads, but it will not run any faster as more threads are added. In fact, this program will slow down as more threads are added because the threads will waste system resources as they wait for their turn to execute the critical section. In essence, the concurrency-control protocol eliminates all of the available concurrency.

The solution to this problem is to take advantage of the fact that the swap operations on the shared data only interfere when at least one of the locations being swapped is the same in both operations. Hence, the right concurrency-control protocol uses pairwise synchronization with nested locks, thereby adding only modest overhead when loop iterations do not interfere. The approach used in [ADE+01] is to create an OpenMP lock for each chromosome. Pseudocode for this solution is shown in Fig. 6. In the resulting program, most of the loop iterations do not actually interfere with each other. The total number of chromosomes, \( N_{\text{POP}} \) (40,000 in the SPEC OMP2001 benchmark), is much larger than the number of UEs, so there is only a slight chance that loop iterations will happen to interfere with another loop iteration.

OpenMP locks are described in the OpenMP appendix, Appendix ???. The locks themselves use an opaque type, \texttt{omp_lock_t}, defined in the \texttt{omp.h} header file. The lock array is defined and later initialized in a separate parallel loop. Once inside the chromosome-swapping loop, the locks are set for the pair of swapping chromosomes, the swap is carried out, and the locks are unset. Nested locks are being used, so the possibility of deadlock must be considered. The solution here is to order the locks using the value of the indices of the array element associated with the lock. Always acquiring locks in this order will prevent deadlock when a pair of loop iterations happen to be swapping the same two elements at the same time. After the more efficient concurrency-control protocol is implemented, the program runs well in parallel.

**Known uses**

A solution to the phylogeny problem described in the Context section is presented in [YWC+96]. The overall approach fits the Task Parallelism pattern; the rejected-sets data structure is explicitly managed using replication and periodic updates to reestablish consistency among copies.
#include <omp.h>
Int const NPOP // number of chromosomes (~40000)
Int const NCHROME // length of each chromosome

Real :: tempScalar
Array of Real :: temp(NCHROME)
Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: fitness(NPOP)
Int :: j, iother

#pragma omp parallel for
loop [j] over NPOP
iother = par_rand(j) // returns random value greater
// than or equal to zero but not
// equal to j and less than NPOP

#pragma omp critical
{
    // Swap Chromosomes
    temp(1:NCHROME) = iparent(1:NCHROME, iother)
    iparent(1:NCHROME, iother) = iparent(1:NCHROME, j)
    iparent(1:NCHROME, j) = temp(1:NCHROME)

    // Swap fitness metrics
    tempScalar = fitness(iother)
    fitness(iother) = fitness(j)
    fitness(j) = tempScalar
}
end loop [j]

Figure 5: Pseudocode for an ineffective approach to parallelizing the population shuffle in the genetic algorithm program GAFORT
0.1 The Shared Data Pattern 15

```
#include <omp.h>
Int const NPOP // number of chromosomes (~40000)
Int const NCHROME // length of each chromosome

Array of omp_lock_t :: lck(NPOP)
Real :: tempScalar
Array of Real :: temp(NCHROME)
Array of Int :: iparent(NCHROME, NPOP)
Array of Int :: fitness(NPOP)
Int :: j, iother

// Initialize the locks
#pragma omp parallel for
for (j=0; j<NPOP; j++){ omp_init_lock (&lck(j)) }
#pragma omp parallel for
for (j=0; j<NPOP; j++){
    iother = par_rand(j) // returns random value >= 0, != j,
    // < NPOP
    if (j < iother) {
        set_omp_lock (lck(j)); set_omp_lock (lck(iother))
    }
    else {
        set_omp_lock (lck(iother)); set_omp_lock (lck(j))
    }

    // Swap Chromosomes
    temp(1:NCHROME) = iparent(1:NCHROME, iother);
    iparent(1:NCHROME, iother) = iparent(1:NCHROME, j);
    iparent(1:NCHROME, j) = temp(1:NCHROME);

    // Swap fitness metrics
    tempScalar = fitness(iother)
    fitness(iother) = fitness(j)
    fitness(j) = tempScalar
    if (j < iother) {
        unset_omp_lock (lck(iother)); unset_omp_lock (lck(j))
    }
    else {
        unset_omp_lock (lck(j)); unset_omp_lock (lck(iother))
    }
} // end loop [j]
```

Figure 6: Pseudocode for a parallelized loop to carry out the population shuffle in the genetic algorithm program GAFORT. This version of the loop uses a separate lock for each chromosome and runs effectively in parallel.
Another problem presented in [YWC+96] is the Gröbner basis program. Omitting most of the details, in this application the computation consists of using pairs of polynomials to generate new polynomials, comparing them against a master set of polynomials, and adding those that are not linear combinations of elements of the master set to the master set (where they are used to generate new pairs). Different pairs can be processed concurrently, so one can define a task for each pair and partition them among UEs. The solution described in [YWC+96] fits the Task Parallelism pattern (with a task queue consisting of pairs of polynomials), plus explicit management of the master set using an application-specific protocol called software caching.

Related Patterns

The Shared Queue, Shared Map, and Distributed Array patterns discuss specific types of shared data structures. Many problems that use the Shared Data pattern use the Task Parallelism pattern for the algorithm structure. The Speculation pattern gives a general discussion of speculation.
Bibliography


