COOL: A Language for Parallel Programming

Rohit Chandra, Anoop Gupta and John L. Hennessy

Technical Report No. CSL-TR-89-396

October 1989

This research has been supported by the Defense Advanced Research Projects Agency under contract No. N00014-87-K-0828.
COOL: A Language for Parallel Programming

Rohit Chandra, Anoop Gupta and John L. Hennessy


October 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

We present COOL, an object-oriented parallel language derived from C++ by adding constructs to specify concurrent execution. We describe the language design, and the facilities for creating parallelism, performing synchronization, and communicating. The parallel construct is parallel functions that execute asynchronously. Synchronization support includes mutex functions and future types. A shared-memory model is assumed for parallel execution, and all communication is through shared-memory. The parallel programming model of COOL has proved useful in several small programs that we have attempted. We present some examples and discuss the primary implementation issues.

Key Words and Phrases: parallel programming, programming languages, C, C++, object-oriented programming, concurrency, shared-memory, synchronization, futures, mutual exclusion, monitors.
1 Introduction

The goal of this project is to design a general-purpose language for parallel processing, that will enable us to use parallel machines effectively. The primary design goals are:

Efficiency: It should be possible to implement the parallel constructs efficiently. The performance gain they provide should more than compensate for any associated overhead. A program should not have to pay for features that it does not use.

Expressiveness: The language should facilitate the construction of a large variety of parallel programming paradigms. This will encourage experimentation with different decompositions of a problem.

The object-oriented approach to parallel processing is promising because object-oriented programs are modular: side-effects may be confined within the class functions executing on an object, and the data dependencies may be made explicit in the interface between an object and its users. This modularity helps the programmer cope with multiple, simultaneously executing threads. The language COOL (Concurrent Object Oriented Language) exploits the object-oriented paradigm for concurrent programming by organizing concurrency around classes.

COOL is an extension of C++ [12], an object-oriented extension of C [9]. Extending an existing language has the advantage of offering concurrency features in a familiar programming environment. We chose C++ because it supports object-oriented programming, is widely used, and compilers for it are freely available. The implementations are efficient and provide a good standard for performance comparisons.

The language COOL is designed to facilitate expression of medium to large grain parallelism. Compilers can automatically extract fine-grain parallelism for architectures that support such a level of concurrency. The significant new features of COOL are:

- parallel fun tions that execute asynchronously when invoked,
- mutex functions that execute atomically on an object, and
- future types that incorporate synchronization as part of the shared object.

The paper discusses these constructs and how they are integrated with C++. The language is evolving, and we do not discuss the interaction of our constructs with the more detailed features of C++ such as friend or virtual functions, or overloading.

Section 2 describes the design of the language and discusses the new constructs in detail. Section 3 presents some example programs. Section 4 discusses the implementation of future types, and other implementation issues. Section 5 compares COOL with other approaches to parallel programming. Section 6 offers some concluding remarks and presents directions for future research.

2 The Language Design

COOL extends C++ with constructs for parallelism. We can classify these constructs as follows:

1. Class based concurrency and synchronization: These constructs specify both concurrency and synchronization at the granularity of functions within a class.
<table>
<thead>
<tr>
<th>Example</th>
<th>What is it?</th>
<th>What does it do?</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel int foo()</td>
<td>Function declaration</td>
<td>Declare foo() to be a parallel function</td>
</tr>
<tr>
<td>mutex int foo()</td>
<td></td>
<td>Declare foo() to be a mutex function</td>
</tr>
<tr>
<td>mutex parallel int foo()</td>
<td></td>
<td>Declare foo() to be a parallel mutex function</td>
</tr>
<tr>
<td>int foo()</td>
<td></td>
<td>Declare foo() to be a sequential, non-mutex function</td>
</tr>
<tr>
<td>parallel foo()</td>
<td>Function invocation</td>
<td>Invoke foo() asynchronously</td>
</tr>
<tr>
<td>parallel- foo()</td>
<td></td>
<td>Invoke foo() synchronously</td>
</tr>
<tr>
<td>foo()</td>
<td></td>
<td>Invoke foo() as specified in the declaration</td>
</tr>
<tr>
<td>int$ x</td>
<td>Variable declaration</td>
<td>Declare x as a future variable of type int</td>
</tr>
<tr>
<td>waitset(x)</td>
<td>synchronize</td>
<td>Block till x resolves</td>
</tr>
<tr>
<td>waitclear</td>
<td>synchronize</td>
<td>Block till x is unresolved</td>
</tr>
<tr>
<td>set(x)</td>
<td>Resolve</td>
<td>Explicitly set x to be resolved</td>
</tr>
<tr>
<td>clear(x)</td>
<td>Unresolve</td>
<td>Explicitly unresolve x</td>
</tr>
<tr>
<td>isset(x)</td>
<td>Test status</td>
<td>Test if x is resolved or not</td>
</tr>
<tr>
<td>blocklock lockB</td>
<td>Lock declaration</td>
<td>Declare lockB of type blocklock</td>
</tr>
<tr>
<td>spinlock lockS</td>
<td>Lock declaration</td>
<td>Declare lockS of type spinlock</td>
</tr>
<tr>
<td>lock(lockvar)</td>
<td>Lock statement</td>
<td>Acquire the lock var</td>
</tr>
<tr>
<td>unlock(lockvar)</td>
<td>Unlock statement</td>
<td>Release the lock var</td>
</tr>
<tr>
<td>release(x)</td>
<td>Release Statement</td>
<td>waitset(x) and release current object</td>
</tr>
<tr>
<td>release(lockB)</td>
<td>Release Statement</td>
<td>lock(lockB) and release current object</td>
</tr>
</tbody>
</table>

Figure 1: Summary of language extensions

(a) **Concurrency**: Functions of a class may be identified as parallel functions in the class definition. Such parallel functions execute asynchronously when invoked. The user of the object need not be aware of the manner of execution; this decision is encapsulated in the class definition. All functions execute in the same address space.

(b) **Synchronization**: Functions may also be identified as mutex functions, signifying that they require exclusive access to an object while executing. This enables synchronization across functions executing on the same object.

2. **Object based synchronization**: COOL introduces new data types called *future* types, which incorporate synchronization for shared data structures as a property of the data itself.

The model of parallel execution is asynchronous functions executing on objects. The decomposition of the problem and design of data structures must be targeted to this model to exploit maximal parallelism.

Figure 1 shows a list of the new constructs added to C++. The rest of this section discusses the mechanisms to create parallelism, synchronize, and communicate.

2.1 **Creating Parallelism**

We create parallelism in a COOL program by the invocation of parallel functions. A member function of a class, either private or public, may be declared to be a parallel function. As with futures in MultiLisp [7], invocations of this function result in an asynchronous function call; the calling thread does not wait for the call to complete but executes concurrently with the invoked function. Synchronization for the result of the call is transparent to the caller, and occurs at the first subsequent access to the result that needs the value. There are some significant differences between futures in MultiLisp, and parallel functions in COOL. MultiLisp futures are associated with the invocation of an expression; in COOL, they are associated with the declaration of a function. Synchronization for the return value of a future in Multilisp is transparent to the programmer and is
implemented (conceptually) through run-time checks on all variables\(^1\). Synchronization for the return value of a parallel call in COOL is provided through variables of type future. These are new data types in COOL and are implemented by the compiler. Section 2.2.2 discusses future types in greater detail.

The COOL model declares parallel functions whose invocations automatically execute asynchronously. The parallelism is thus encapsulated as an implementation detail within the class definition, and the user of an object need not be aware of the underlying parallel execution. This is the preferred way of using parallel functions. However, there are occasions when we want to specify the manner of execution at an invocation site. For instance, asynchronous invocation is desirable only if the caller has useful computation to perform while the result is being computed. It is otherwise futile to pay the overhead of creating a parallel task and have the caller sit idle. In such situations the caller may insist on sequential execution by specifying the parallel-attribute at an invocation site. Similarly, invocation with the parallel attribute insists on concurrent execution. Thus an attribute specified at an invocation site overrides the default specified in the function declaration. Invocations without an attribute execute in the default manner specified in the function declaration.

The forms of class-based concurrency possible in COOL are:

1. **Concurrency Across Objects**: Functions on different objects may execute concurrently.
2. **Concurrency Within an Object, Across Functions**: Several functions may execute simultaneously on the same object.
3. **Concurrency Within a Function**: A function executing on an object may invoke other parallel functions.

### 2.2 Synchronization

Synchronization support in COOL includes mutex functions that execute exclusively on an object. They provide synchronization at the function level on an object. Future types enable synchronization at a finer granularity than functions, and help exploit concurrency from parallel function invocations. Finally, locks are provided for efficient fine-grain synchronization.

#### 2.2.1 Synchronization with mutex functions

We specify the mutex attribute for member functions of a class similarly to the parallel attribute. A mutex function requires exclusive access to the object it is invoked on; no other public member function may execute concurrently on the object. A mutex function cannot start executing until all other functions executing on the object have completed; other function invocations on the object cannot start until the mutex function completes. Functions are assumed by default not to require exclusive access, and several of them may execute simultaneously on an object so long as no mutex function is running.

The mutex attribute provides multiple-reader single-writer style access to data objects. The reader functions do not require exclusive access and can execute concurrently with other reader functions. However, a function which needs write access cannot execute concurrently with other operations and must be declared as a mutex function.

\(^1\)A compiler may optimize away many of these run-time checks.
An executing mutex function foo may need to wait for an event that is caused by another function bar on the same object. If foo blocks waiting for the event then bar will never execute on the object since it is locked by foo, resulting in deadlock. In such situations the release statement allows a function to atomically block itself on an event and release the object for other functions. When bar causes the desired event foo resumes execution at the point it left off, after any executing functions complete. This ensures that only one function executes on the object at any time. Once the desired event has occurred other functions invoked on the object are blocked till foo resumes and exits, releasing the object. If the event causes several waiting functions to be woken-up then they resume execution one at a time, again maintaining a single mutex function within the object at any time. The release () statement must be invoked from within a member function that is declared to bemutex.

The mutex facility is similar to the monitor construct [8] in that it incorporates synchronization as a property of the object. However, monitors are stricter than mutex since all functions require exclusive access to the object while executing. With the mutex facility access to an object is more flexible than in monitors, and may enable additional concurrency to be exploited, as illustrated in the hash table example of Section 3.1. The release statement is similar to waiting on a conditional variable in a monitor. It is more flexible in that a function can block on either future variables or locks (discussed later in Sections 2.2.2 and 2.2.3 respectively), or choose to block without releasing the object.

Some properties of the mutex attribute are worthy of note. First, mutex and parallel are orthogonal attributes, and each can be specified for a function independent of the other. Thus a function may have both the mutex and parallel attributes specified for it. Second, the mutex attribute provides synchronization at the function level. Synchronization at a granularity finer than task or function granularity is possible through future types. Third, synchronization is against other functions on the same object; functions on other instances of the same or another class may execute concurrently. Fourth, synchronization is against other public member functions; private functions may be invoked from within an executing mutex function without deadlock. Finally, it provides synchronization only against other member functions. In C++ (and hence in COOL) the public fields of an object may be accessed directly while a mutex function is executing. This may result in improved performance, as discussed in Section 2.3. Synchronization in such cases must be managed by the programmer using future types or locks, which are discussed next.

### 2.2.2 Synchronization with future types

Future types are new data types in COOL. They consist of a base type that may either be a primitive type of C++, a user defined type or a class. Future types are specified by appending the base type with the $ symbol (e.g. int $).

A variable of type future (future variable) is accessed as if it were a variable of the base type. It can be bound to all possible values of the base type. A future variable differs from a base type variable in that it can be unresolved. This signifies that its value is not available now, but will be available later. Access to an unresolved future variable blocks automatically until the value is determined. Access to a resolved future variable behaves like one to a base type variable.

Some operations on future variables refer to the value but don’t actually need it. Simple assignment and parameter passing only need to reference the name and do not need to wait for the value to be determined, resulting in additional concurrency. These are non-strict operations. However, all other operations (like addition and comparison) need the value and must block until it is determined. These are strict operations.

As stated above, the base type of a future type may be a primitive type, a user-defined type, or
a class. The first two have straightforward semantics since the variable only has a value. If the base type is a class then the variable (object) has the additional property that operations may be invoked on it. An operation invoked on a future object is allowed to continue only if the object is resolved. If the object is unresolved then the operation blocks, to be awoken when the object is determined.

We discuss the properties of future variables below.

**Behavior of Future Types and their Relation to the Base Type:**
A future variable may be used at any site where a value of its base type is required. For instance, if \( qx \) is of type \( \texttt{int} \) then it may be used in the expression \( qx+2 \). This uses blocks till \( qx \) is resolved and its value determined. The value is implicitly type converted to one of the base type before being used. Similarly, a value of the base type may be used at any site where one of the future type is expected. Its value is automatically coerced to a future type and used. For instance in the assignment \( qx=3 \) the integer 3 is coerced to a \( \texttt{int} \) with the determined value 3 and assigned to \( qx \). Thus base types and future types may be used interchangeably with the implicit type conversion rules of COOL.

**Non-strict Access to Future Variables:**
Simple assignment and passing as a parameter are the two instances of non-strict access to future variables. They do not need to wait for the future to be resolved. Consider the assignment \( qx=qy \) where both \( qx \) and \( qy \) are future variables (the parameter passing case is similar). This statement is similar to an assignment statement of regular variables: \( qx \) receives the value of \( qy \). Its effect is to atomically copy the value that \( qy \) has, or is expecting, into \( qx \). Thus the assignment is by value, and subsequent changes to one do not affect the other. If \( qy \) is resolved then \( qx \) is resolved to the value of \( qy \), and subsequent changes to one do not affect the other. \( qy \) may be expecting a value in two ways. If \( qy \) is unresolved awaiting the result of a parallel function then \( qx \) becomes unresolved awaiting that value as well. Subsequent assignments to one do not affect the other. The second possibility is if \( qy \) is unresolved but not awaiting any function return value; then \( qx \) becomes unresolved awaiting the assignment of a resolved value to \( qy \). after which the variables \( qx \) and \( qy \) are independent.

**The set () statement:**
As mentioned above assignment to a determined value resolves a future variable. The statement \( \text{set}(qx) \) also sets \( qx \) to be resolved. The set () statement serves two functions. First, future variables are often used for synchronization where only their state (whether resolved or unresolved) is useful. With set () they can be explicitly resolved without having to assign them a dummy determined value. The second need for the set () statement arises if the base type of \( qx \) is a structured type. The various sub-fields of \( qx \) may need to be specified separately, through several assignments. Partial assignments to fields of a complex structure do not resolve a structure. This ensures that the structure is not resolved while only some fields have been specified. After specifying all the values the structure may be resolved with an explicit set () statement. A future variable of a complex base type may be set to resolved by either a structure assignment to a determined structure or the set () statement. A future variable of simple type may be set to resolved by either assignment to a determined value or the set () statement.

**The isset () expression:**
The expression is set \((qx)\) is a non-blocking call to examine the status of \( qx \). It returns 1 (true) if \( qx \) is resolved, 0 (false) otherwise. It is used to examine the status of a future value, maybe the return value of a previously invoked parallel function or a value being produced by some other concurrently executing function. This is useful for non-deterministic execution as illustrated in the last example in Figure 2.
The release ( ) statement:
The release (qx) statement (see Section 2.2.1) must be invoked from within a mutex function. It is similar to wait set (qx) in that it blocks for qx if qx is unresolved, and has no effect otherwise. It differs from wait set in that if qx is unresolved then it atomically releases the object locked by the mutex function so that other functions can be invoked on it.

Future Types and parallel functions:
Future types help exploit the concurrency of parallel functions. A parallel function executes asynchronously when invoked. If it has a non void return value, then the return value may be declared to be a future type and assigned to a future variable in the caller. The future variable acts as a placeholder for the value which the parallel function is computing. Synchronization in the caller for the return value is delayed until a subsequent strict access to the future variable, when the value is actually required. This permits maximal concurrency between the caller and the invoked function. Being assigned the future return value of a parallel function makes a future variable unresolved. When the parallel function completes, it transparently resolves the variable to the determined return value. All waiting accesses to the variable may then continue with the value.

Future types may also be used as the return type of a sequential function bar ( ). Invocations of bar execute sequentially but return a future value. This is useful if bar invokes a parallel function f oo ( ); it need not block for f oo to complete and can return the future value returned by f oo to its (bar's) invoker.

A parallel function always executes in parallel. However, the type of the return value and the manner of invocation determine if an invocation can continue without waiting for the function to complete. For instance, consider a parallel function foo ( ) having a future return type but invoked with the return value being assigned to a non-future variable. Although f oo executes concurrently with the caller, assignment to a non-future variable is a strict access to the future return value and the caller blocks while f oo executes and the return value is determined. Thus such an invocation results in synchronous execution.

Next consider a parallel function f oo ( ) defined with a non-future return type and invoked with the return value being assigned to a future variable qx. Invocations of f oo execute concurrently and the caller does not block. The future variable qx remains unresolved until f oo completes, whereupon f oo's return value of the base type is coerced to a future value and used to resolve qx.

The advantages of future types as synchronization mechanisms are:

1. The shared data object is the unit of synchronization. Synchronization is an integral property of the data, rather than an external unrelated mechanism.

2. Synchronization is largely transparent with automatic block on access until the value is determined.

3. Synchronization granularity is separated from task granularity, and can be at a finer grain. Tasks can synchronize at intermediate points during computation, rather than only at entry and exit, as is possible with the mutex facility.

This is in contrast to Multilisp where futures provide synchronization only at function call boundaries. Besides synchronizing for parallel functions future types in COOL are useful as general synchronization mechanisms.
Future types are similar to promises [11] in that both are strongly typed and implemented by the compiler. A future variable (promise) can be bound to the return value of parallel functions (asynchronous methods). The value of a future object (promise) does not change once it is determined, and changes to a future variable require a new future object. The compiler/run-time system may optimize this where possible without affecting the semantics. Promises and future types differ in that a promise must be explicitly claimed (synchronized for) by the caller. This synchronization is automatically done on accesses to future variables in COOL. Future types may be used for more flexible synchronization patterns besides the return value of asynchronous function invocations. Promises implement exceptions as they were designed to cope with communication errors in a distributed system.

Figure 1 lists the various constructs of COOL including those that manipulate future types. Figure 2 illustrates how parallel functions are defined and invoked, and presents a few small examples that manipulate future variables and demonstrate the type casting.

2.2.3 Synchronization with locks

The mutex attribute is useful for synchronizing access to an object through member functions. Future types integrate synchronization with the shared data objects. Locks may be required for synchronization at a finer granularity, when necessary for reasons of efficiency. COOL provides two kinds of locks to synchronize access to variables shared across concurrently executing threads. The primitives available are:

```plaintext
blocklock Lvar
spinlock Lvar

Declare Lvar to be a lock variable, either a blocklock or a spinlock.
```

```plaintext
lock(Lvar)

The lock function attempts to acquire the lock Lvar. The calling thread waits for the lock to become available, locks it, and continues. A blocklock causes the waiting process to be suspended and placed on a queue of processes waiting for the lock. A spinlock busy waits on the lock until it is available. A blocklock involves the overhead of blocking a process and a context switch, while a spinlock consumes resources during busy-wait. A blocklock is preferable if the waiting time for a lock is expected to be larger than the context switch overhead.
```

```plaintext
unlock(Lvar)

The unlock function unlocks the lock, wakes up the first waiting process in the case of a blocklock, and continues.
```

```plaintext
release(Lvar)

Lvar must be a blocklock and the statement must be within a mutex function. As discussed in Section 2.2.1, this statement blocks till the lock Lvar is available. If the lock is not immediately available then the object is released.
```

2.3 Communication

COOL does not provide any explicit constructs for communication between multiple tasks. Since COOL assumes a shared-memory model, the facilities available in C++ for communication across function calls enable communication across tasks in COOL. Communication is possible through function arguments and return values, through global variables uniformly visible in all functions, and through public variables of a class. Passing pointers to data permits efficient sharing of data. The
class testclass {
    ...;
public:
    parallel int$ foo(); // Declare foo to be a parallel function
};
main() {
    testclass obj;
    int$ qx, qy; // future variables
    int i, j; // ordinary variables

    qx = 3; // The integer 3 is type cast to a future int.
          // qx becomes resolved with the value 3.

    qx = obj.foo(); // Asynchronous invocation of foo.
                    // qx becomes unresolved, awaiting
                    // completion of foo().

    i = j + qx; // strict access to qx.
                  // Block till the value is determined

    qx = obj.foo(); // asynchronous call again

    qy = qx; // non-strict access to qx, non-blocking
              // qy becomes unresolved, awaiting the same
              // value as qx

    i = qy; // Since i is not a future variable, this
             // assignment blocks till qy is resolved.
              // The value is then cast to an int,
              // and assigned to i
              // Both qx and qy are now resolved with
              // the same value

    j = obj.foo(); // Since j is not a future variable, this is
                   // a strict access to the return value of foo
                   // Illustrates how (int$) is cast to int

    qx = obj.foo(); // asynchronous call
    ...
    if isset(qx) // examine qx to see if resolved or not
       { ... } // use the new value
    else { ... } // use the old value
    // results in a non-deterministic choice.
    ...
    waitset(qx); // explicitly synchronize.
    ... // Block till qx is resolved
    ...}

Figure 2: Example: Parallel Functions and Future Types
shared-memory model ensures that references from different tasks to global variables refer to the same location.

The public variables of a class are accessible directly without having to invoke a member function. This violates the modularity of objects and side-effects are no longer confined within functions. Violating modularity destroys the mutex facility which provides synchronization only across functions, and can lead to subtle bugs. In such cases synchronization must be managed by the programmer using locks. However, modular code can sometimes be restrictive in both expressiveness and efficiency. Allowing direct access to some fields of an object may save unnecessary copying of data and the overhead of a function call. This can often result in substantial performance improvement as demonstrated by the merge sort program in Section 3.2.

3 Example Programs

The object-oriented model provides a useful way to organize concurrency and synchronization. We have written several programs in COOL including the bounded-buffer problem, an LU decomposition using Gaussian elimination, the simplex algorithm for linear programming, the travelling salesman problem, and the maximum flow in a graph (maxflow) problem. In this section we present two small examples that illustrate the use of the parallel features of COOL. They are a concurrent hash-table implementation, and a merge-sort algorithm.

3.1 Concurrent Hash Table

The first example is an implementation of a concurrent hash table. It illustrates how data can be organized to permit parallel operations, and how the mutex attribute facilitates the multiple-reader single-writer paradigm of concurrent computation. Records are stored in a list on which three operations can be performed: lookup, insert, and remove. This list is partitioned into buckets. The hash function uses the key value of a record to determine its bucket number. Since a lookup is a read-only operation, various lookups can execute concurrently on either the same or different buckets. Insertion and removal require exclusive access to the bucket because they modify the data. However, insert and remove operations for different buckets can execute concurrently.

The code for the example is shown in Figure 3. The class Hashtable defines the interface: the public functions are insert, remove, and lookup. The hash function is private; it takes the record key and returns the bucket number in which the record may be found. HashTable’s private data consists of an array of buckets. All three public functions are parallel functions and do not block the user of the hash table. Each lookup, insert or remove call will spawn a parallel task. The user blocks only when the value of a fetched record is accessed. None of the functions are mutex functions; exclusive access is only required within a bucket.

The class BucketType consists of an array of records, and provides the insert, remove, and lookup operations. Insert and remove are mutex functions, since they require exclusive access to the bucket. The lookup operation is read-only and does not require exclusive access. Operations on a Bucket Type perform the real work of fetching, inserting, or removing a record. Insertion and removal require exclusive access to the bucket, but several lookup operations may execute concurrently on the same bucket. Since a parallel thread to perform the requested operation has already been spawned by the HashTable class, no functions need be declared parallel functions.
class HashTable {
    BucketType buckets[MaxBuckets];
    int hash(KeyType);
public:
    parallel void insert(KeyType, RecordType);
    parallel void remove(KeyType);
    parallel RecordType$ lookup(KeyType);
};
parallel int HashTable::insert(KeyType key, RecordType record)
{
    buckets[hash(key)].insert(key, record);
}
parallel int HashTable::remove(KeyType key)
{
    buckets[hash(key)].remove(key);
}
parallel RecordType$ HashTable::lookup(KeyType key)
{
    return(buckets[hash(key)].lookup(key));
}
int HashTable::hash(KeyType key)
    // ...return bucket number that the key hashes to...
}

class BucketType {
    RecordType list[MaxSize];
public:
    mutex void insert(KeyType, RecordType);
    mutex void remove(KeyType);
    RecordType lookup(KeyType);
};
mutex void BucketType::insert(KeyType key, RecordType record)
{
    // ...find index in bucket to insert...
    list[index] = record;
    // ...move other records around if need be...
}
mutex void BucketType::remove(KeyType key)
{
    // ...find index in bucket to remove...
    list[index] = NULL;
    // ...move other records around if need be...
}
RecordType BucketType::lookup(KeyType key)
{
    // ...find index in bucket to return...
    return(list[index]);
}

Figure 3: Example: A Hash Table
```cpp
class array {
    int count, *aptr; // count = number of elements.
    // aptr = pointer to list of elements.
public:
    array(int, int*);
    parallel int$ sort();
};
array::array(int n, int *s)
    count = n;
aptr = s;
parallel int$ array::sort()
    if (count<MinSize) {
        ...size too small to divide: sort using a serial algorithm...
    }
    else {
        array left(count/2, aptr);
aptr = s;array right(count - (count/2), aptr+count/2);
        // create and initialize the left and right halves
        int$ done = left.sort();
        // invoke sort on the left half in parallel
        parallel- right.sort();
        // invoke sort on the right half sequentially.
        // overlap sorting of the two halves
        waitset(done); // synchronize for the left half to get sorted
        ...now serially merge the two sorted halves...
    }
}
```

Figure 4: Example: Merge Sort

### 3.2 Merge-Sort

The second example implements a merge-sort based on the divide-and-conquer paradigm. The list to be sorted is split into two approximately equal parts, each of which is sorted separately. The sorted halves are then merged. Sorting the left and right halves are independent tasks and can be done concurrently.

The class `array` in Figure 4 defines the `sort()` function. The variable `aptr` points to the list of numbers to be sorted. If there are fewer than `MinSize` integers in the list, then some serial algorithm is used. Otherwise `sort()` creates two array objects and initializes them with pointers to the left and right halves of the current list. Note that since the split is done using pointers, no elements need to be copied. The left half of the list is sorted in parallel with the right half by invoking `sort()` on the left half in parallel, and sorting the right half sequentially in the caller. The invocation for `sort()` on the right half has the parallel- attribute, demanding sequential execution. The merge part is done sequentially once both halves have been sorted.

This example illustrates how to overlap concurrent tasks and synchronize when results are required. The shared-memory model saves the cost of repeatedly copying elements of the list across tasks.
4 Implementation Issues

The programmer’s view in COOL is that of an unlimited number of homogeneous processors all sharing the same address space. Parallel execution is obtained through asynchronous function calls, without requiring (of the programmer) creation of a task, moving data to shared memory, allocating resources, or scheduling. All this is done transparently by the implementation. The only difference between synchronous and asynchronous calls is in concurrent execution, and in the accompanying non-determinism and possible violation of data dependencies. Parallel threads execute in the same address space, similar to the sequential case.

The major implementation issues are the implementation of future types, the run-time environment, memory management, and implementation of scope rules.

4.1 Implementation of Future Types

The implementation of future types requires that future variables be possibly bound to an unresolved value. Strict accesses to future variables wait until the value is determined. Since future variables are known at compile time the compiler can generate code ensuring that all accesses to future variables check the state of the variable. Run time status information needs to be maintained for each future variable; therefore there is a structure associated with each future variable. Typical information in such a structure would include:

- status of the variable, which may be resolved or unresolved.
- value of the variable, a field of the base type. It contains the actual value of the variable if the status is resolved.
- queue on which an access waits if the variable is unresolved. Waiting accesses continue when the value becomes determined.

The three possible states of a future variable are reserved, clear and set. A variable is reserved if it is unresolved, associated with an executing parallel function, and awaiting its return value. A variable is clear if it is unresolved, but there is no corresponding parallel function. This is the case when a variable is initially declared, or is explicitly cleared with the clear() statement. As shall become clear below, we need to distinguish between these two states of a variable. A variable is set if it has a determined value. An unresolved variable is one which is either reserved or clear. A resolved variable is one which is set.

We define a blocked access to be one which attempted to access an unresolved (either clear or reserved) variable. Therefore it has to block until the variable is resolved. To define a child variable consider the statement qx=qy, where both qx and qy are future variables. If qy is unresolved, then qx inherits the state of qy, waiting for the value of qy to be determined. When qy gets resolved, qx should get the determined value as well. Thus qx is said to be a child of qy. However, the relationship is not symmetric; qy is not a child of qx, since changes to qx do not affect qy. For instance, a subsequent assignment to qx, say qx=3, will not affect qy. If qy is resolved then qx is assigned its value and does not become a child of qy.

A variable which is set will not have any blocked accesses or child variables. A reserved or clear variable may have either or both of these.

Let qx and qy be future variables, and z be a regular variable. We discuss the implementation in terms of the behaviour of the future variable qx. We may use a future variable in two different situations. First, where a value of the base type is expected (e.g. z=qx+1). If qx is unresolved then
the access must block until the value becomes available. If it is set then the value is used without blocking. Second, we may use qx where a value of the future type is expected (e.g. qy=qx). This access is non-blocking even if qx is unresolved. However, if qx is reserved or clear then qy becomes a child of qx. The parallel function computing qx, or any other subsequent assignment to qx also updates qy.

Future variables may be assigned to in several ways. Statements may alter the state of the variable to become set, reserved, or clear. We consider each of these possibilities in turn.

Statements that set a variable (e.g. set (qx) or qx=23) set the variable to a determined value with no blocked accesses or child variables. Figure 5 illustrates their effect. They may find q-x in any of the three states when they execute. If qx was set then only its value needs to be changed. If qx was reserved these continue to await the return value of the associated parallel function. If it was clear then they receive the value being assigned to qx.

Statements that reserve a variable (e.g. qx=parallel obj.foo( . . . )) make qx unresolved awaiting the value of an executing parallel function. They are presented in Figure 6. The case when qx was previously set is straightforward. If qx was reserved then any blocked accesses and child variables receive their value from the previous function obj . j . bar ( ) . If it was clear then they receive their value from the new function, obj . j . f oo () .

Finally, statements may clear a variable (e.g. clear (qx) ) and are illustrated in Figure 7. If qx was set then is cleared. If it was reserved then the blocked accesses and child variables receive their value from the executing parallel function obj . j . f oo () . If it was clear then there is no change.
/********** previous state set **********/
xq = 3;  // qx is set, value = 3
...
qx = parallel obj.foo();  // qx becomes reserved, awaiting the
// value of the parallel function foo()

/*******XX previous state reserved ******I
qx = parallel obj.bar();  // qx is reserved, awaiting the
// value of the parallel function bar()

/*******XX previous state clear ******I
qx = parallel obj.foo();  // qx becomes reserved, awaiting the
// return value of obj.foo()
// Blocked accesses and child variables
// receive their value from obj.bar()

clear(qx);  // qx is clear
...
// It may acquire blocked accesses and
// child variables
...
qx = parallel obj.foo();  // qx becomes reserved, awaiting the
// value of obj.foo()
// Blocked accesses and child variables
// receive their value from obj.foo()

Figure 6: Example: Effect of statements that reserve a future variable qx

With these general requirements of an implementation, we outline two possible schemes. The first
is to implement a future variable as a record structure that includes the value field and other necessary
fields. Each future variable has its own structure; they are not shared. Thus variables may be updated
in place. However, it becomes harder to keep track of child variables and blocked accesses since child
variables have their own structure. The salient aspects of the algorithm are:

1. The future function keeps a list of all structures that need the value it is computing.
2. An unresolved structure keeps a list of its child variables which must be updated when it gets
   resolved.
3. Each structure has a pointer to the parallel function or parent variable (if any) that it is
going to get its value from. If, in the meanwhile, this variable gets set by some other means
then it should remove itself from the list of the parallel function or parent variable.

Another possibility is to implement a future variable as a pointer to a structure. It is not neces-
sary to maintain lists of structures that need to be updated; since structures are shared this happens
automatically. However, since several variables may share a structure, a new structure will have to
created on every update to a variable. This may be optimized by maintaining reference counts of
the number of variables which are sharing a structure. It is important to maintain which variable is
the owner of the structure, and which ones are children of that variable. Only changes to the owner
variable should be reflected in the structure, not those to the child variables.
qx = 3;  // qx is set, value = 3
...
clear(qx);  // qx becomes clear

qx = parallel obj.foo();  // qx is reserved, expecting the value
// of the parallel function foo()
...
// qx may acquire blocked accesses
// and child variables
...
clear(qx);  // qx becomes clear
// Blocked accesses and child variables
// receive their value from obj.foo()

qx = parallel obj.foo();  // qx is reserved, expecting the value
// of the parallel function foo()
...
// qx may acquire blocked accesses
// and child variables
...
clear(qx);  // qx becomes clear
// Blocked accesses and child variables
// receive their value from obj.foo()

qx = parallel obj.foo();  // qx is reserved, expecting the value
// of the parallel function foo()
...
// qx may acquire blocked accesses
// and child variables
...
clear(qx);  // qx becomes clear
// Blocked accesses and child variables
// receive their value from obj.foo()

Figure 7: Example: Effect of statements that clear a future variable qx

The advantages of sharing are that all blocked accesses sleep on the same template. Copies are cheaper to make, since they involve only a pointer copy rather than that of the whole structure. Storage sharing saves on the amount of memory consumed, important in a system with no garbage collection.

4.2 Run-time Environment

The primary responsibility of the run-time environment is to handle dynamic task creation. When a parallel function is invoked, a new task must be created, allocated resources, and scheduled to run.

Creating a new process has a large overhead cost. A major portion of this cost is in copying the address space, which we do not require since we allocate all data in shared memory. If every parallel function invocation resulted in a new process, the cost of process creation would make parallel functions useful only for very large granularity. To keep this cost as low as possible, the COOL environment does its own task management. When a parallel function is invoked, a task is created and placed on a task queue. Server processes are created at start-up time, usually one per processor. They pick tasks off the queue in FIFO order and execute them. The queue is globally shared between the servers. Tasks are light-weight since the only resource they require is a stack.

We may later explore more sophisticated techniques for scheduling and allocation. These could include dynamic control over the amount of parallelism, allowing user-specified priorities in the scheduling of tasks, executing a task on a particular processor or set of processors, co-locating certain tasks on the same processor, and creating more than one server per processor.
4.3 Memory Management

The COOL memory management system ensures that all tasks (parallel function calls) execute in the same address space. All local variables in a function are allocated in the activation record for the function. All free variable references should refer to the same location in memory whether the function was invoked sequentially or concurrently. This location should be lexically obvious from the sequential program. Since C++ allows pointers to local variables to be passed across functions, all data needs to be allocated in shared memory.

We discuss below the implementation considerations of the different kinds of variables accessible to a function in COOL.

Global Variables: All global variables must be allocated in shared memory.

Local Variables: Local variables are allocated on the stack associated with a task.

Actual Parameters: Those parameters which are passed by value are allocated on the called functions activation record. However, if a pointer to a variable within another task’s scope is passed as an actual argument, then the stack of the other task should be mapped into this function’s address space. Thus all stack space associated with a task should be allocated in shared memory.

Instance Variables: These variables are accessed through the inheritance hierarchy. Since objects are allocated on the heap, it suffices to map the heap into the shared address space.

4.4 Scope

A potential problem arises when local variables within a function are accessed by a parallel function. This may happen if the first function invokes the second with a pointer to a local variable as an argument. The caller may exit and deallocate its local variables before the called function completes. Any further references to the variables by the still executing called function will encounter garbage.

The initial language design ignores this issue, placing the responsibility on the programmer. To ensure that an invoked parallel function has completed before exiting the caller’s scope, the return value of the function must be explicitly touched in the caller with the wait set statement. This will block the caller till the invoked function has completed. If the called function does not access the calling thread’s scope, then the caller need not wait for it to complete.

This scheme is efficient, since an invoked parallel function is synchronized only if necessary. However, forgetting to wait for a function can lead to bugs that may be very difficult to detect. We plan to explore the following alternative ways of handling this problem.

1. Require all parallel function invoked from within a scope to complete before the caller exits. This scheme ensures secure and correct operation, but may result in reduced efficiency because of unnecessary blocking.

2. Require the programmer (or compiler) to specify one of two kinds of exits from a scope, strict and non-strict. A strict exit would require all parallel function invocations to complete; a non-strict would be an immediate, non-blocking exit that does not wait for any executing tasks to complete.

\[^2\text{Not to be confused with strict and non-strict accesses to future variables in Section 2.2.2.}\]
3. Allow the calling thread’s scope to exit, but keep its stack around until all parallel function
invocations from within that scope complete. This would ensure correct operation when the
tasks access any variable within the scope. However, maintaining the scope after it is exited
may be both difficult and expensive, since extra work would be required at run-time.

5 Related Research

COOL is based on earlier work done at AT&T Bell Labs which is discussed in [2]. Other related
research includes the PRESTO project at the University of Washington and research on concurrent
C++ at Brown University. Both projects attempt to exploit the object-oriented model of C++ for
concurrency. We compare these projects with COOL in some detail.

PRESTO [1] supports parallel programming within C++ through a set of pre-defined classes. To
specify parallel execution, the programmer creates an instance of the thread class and starts it
executing by giving it an object, a function, and the arguments with which to invoke the function. All
threads execute in the same address space; each has a program counter and a stack. The user of an
object can decide whether to invoke a function synchronously (as in C++) or asynchronously with a
thread.

PRESTO and COOL are similar in that both attempt to exploit the object model for concurrency.
Several threads may execute concurrently on an object. The major differences between COOL and
PRESTO are:

1. In PRESTO the programmer has to explicitly create threads; the COOL conceptual model is
simpler since task creation and management is transparent to the programmer.
2. COOL attempts to encapsulate parallelism within the implementation of a class (i.e. in the class
definition). In PRESTO the user of an object decides between synchronous and asynchronous
invocations.
3. There is no automatic synchronization for the return value of an asynchronously invoked function
in PRESTO.
4. PRESTO provides monitors, but the mutex facility of COOL is more flexible (see Section
2.2.1).

Researchers at Brown University have integrated C++ with a threads package [5] to exploit con-
currency on multiprocessors [4]. It provides a predefined class task on which only the constructor
can be invoked. This constructor executes asynchronously and in the same address space. Synchron-
ization for the return value is provided through the task function result (), which blocks till
the value is available. Additional synchronization support includes a queue associated with each task;
processes can wait on this queue till some other process does a wakeup (). A monitor class with
condition variables for synchronization, and dynamically reconfigurable queues for communication are
provided. COOL differs in the following ways:

1. The C++/threads package does not attempt to exploit class-based concurrency. Instead the pro-
gress deals with parallelism at the level of tasks using primitive classes to create parallelism,
to synchronize, and to communicate across tasks. COOL provides constructs fully integrated
with the class-model of C++.
2. Monitors in C++/threads are strict whereas COOL allows the constraints to be relaxed (see
Section 2.2.1).
The ARGUS system [10] is designed for distributed programming. ARGUS Guardians are similar to COOL objects in that they can be accessed through handlers (public functions). Several handlers may execute concurrently if invoked asynchronously. Promises (future types) help implement asynchronous handler invocations (invocations of parallel functions). Several threads (member functions) may execute concurrently within a guardian (object) in the same address space.

In ARGUS the manner of invocation (synchronous or asynchronous) is determined at the invocation site, depending on whether the return value is assigned to a promise or not. In COOL this is a property of the function declaration in the class. In COOL synchronization across different functions may be obtained through the mutex attribute; ARGUS has no such facility and all asynchronous invocations execute concurrently. The ARGUS coenter facility expresses concurrency within a thread, similar to invoking a parallel function from within a concurrently executing function. However, the synchronization in coenter is strict: all sub-tasks within the coenter must complete before the thread is allowed to continue.

Other approaches to exploiting object-oriented concurrency include several variations of Smalltalk [6]. The general approach is to exploit asynchronous message sends for concurrency, similar to invocations of parallel functions in COOL. However, Smalltalk implementations have the overhead of dynamic typing. Information must be maintained at runtime to determine a variable’s type, which can change dynamically. The procedure to be invoked at a procedure call is determined dynamically, based on the type of its first parameter. Garbage collection of objects that are no longer required is expensive.

C++ is strongly typed and more information is available at compile time. Its implementations are correspondingly more efficient than those of Smalltalk. The dynamic nature of programming in Smalltalk was of less importance to us than the efficiency possible with C++.

COOL differs from various Smalltalk extensions in that they require specification of synchronous or asynchronous execution at the invocation site. We briefly review some of the extensions of Smalltalk below.

ConcurrentSmalltalk [14] allows messages to be sent both synchronously and asynchronously. Synchronization for the reply is through an object called CBox, which is similar to future types. ConcurrentSmalltalk also provides atomic objects for synchronizing accesses at the object level. Atomic objects are a stricter version of the mutex facility of COOL.

CST [3] defines all messages as asynchronous, and synchronization for their return values is automatic. Several methods may execute concurrently on an object, but synchronization across tasks is managed by the programmer using locks and semaphores. CST introduces the concept of a distributed object with a single name but distributed state. Making all messages asynchronous relieves the programmer from identifying parallelism, and helps in extracting fine-grain concurrency.

ABCL/I [15] Although it is not an extension of Smalltalk, it is based on communicating autonomous objects that can process only one message at a time. Concurrency across objects is exploited through three types of messages: asynchronous, synchronous, and future.

6 Conclusions and Directions for Future Research

We have described the design of COOL, a concurrent object-oriented language. COOL extends C++ with high-level abstractions that apply the object-oriented approach to parallel programming.
While the effectiveness of object-based concurrency and future types has yet to be demonstrated, our initial experience has been encouraging. The expressiveness of COOL has facilitated useful parallel decompositions of several problems. We hope to report on our programming experience with COOL at a later date.

The primary ideas in COOL are:

1. It introduces concurrency and synchronization support at the function level on objects. This approach integrates well with the object model.
2. It introduces future types as a general synchronization mechanism.

Other interesting features of COOL are that parallelism is encapsulated as part of the implementation of a class, transparent to its users. In other languages the choice of parallel execution is left to the users of the class, made when invoking the parallel function. The mutex feature provides synchronization at the function level. This affords the programmer greater flexibility than synchronization at the object level as in monitors. It exploits class-based concurrency in C++ and hence does not pay the overhead costs associated with dynamic typing. Finally, COOL provides a modular object-oriented model of parallel execution for the security concerned programmer, it simultaneously permits the efficiency concerned programmer to relax some modularity constraints and exploit them for increased performance.

Current and Future Research: We are currently writing application programs in COOL to further understand and refine the language. We plan to implement the language on a shared-memory multiprocessor (the Encore Multimax). An implementation will help us evaluate the performance issues in the language. The implementation has several issues, including dynamic load balancing, scheduling and queuing strategies, controlling the nature and degree of parallelism, and minimizing the overheads due to the concurrency constructs. These are especially interesting in a multi-user multiprocessing environment [13].

Future directions for research include exploiting futures as general synchronization objects. Also, at various stages of designing the language we opted for the simple approach of requiring the programmer to make decisions about the degree and nature of parallelism, scope management, or future variables. Several of those tasks should be done by the compiler or run-time system. Finally, there should be a precise definition of the interaction of the extensions in COOL with features of C++ such as inheritance, virtual and friend functions, and operators.

References


---

3Multimax is a trade-mark of the Encore Computer Corporation.


Acknowledgements: This research has been sponsored by the Defense Advanced Research Projects Agency under DARPA contract #N00014-87-K-0828. Several initial ideas for this research were developed when the first author was visiting the Computing Systems Architecture department at AT&T Bell Laboratories, Holmdel, NJ. Bruce Hillyer and Tom London of AT&T Bell Laboratories contributed several ideas, and their support is gratefully acknowledged. The name for the language, COOL, was suggested by Ashok Subramanian. Comments offered by Paul Calder, Craig Chambers, Kourosch Gharachorloo, Monica Lam, J. P. Singh, as well as the anonymous referees are gratefully acknowledged.