ICC++ Language Definition

Andrew A. Chien and Uday S. Reddy[1]

May 25, 1995

Preface

ICC++ is a new dialect of C++ designed to support the writing of both sequential and parallel programs. Because of the significant investment of programmers and vendors in language skills and tools in C++, one of the major goals is to minimize the differences with C++. Consequently, this document focuses on the essential elements of ICC++ and how they differ from C++. We describe the concurrency model, object collections (an extension of arrays), and finally discuss the programming restrictions which are recommended to increase the effectiveness of program optimization. In general, the reader should assume that C++ features not discussed are incorporated.

This description is not a complete language manual; it is intended to convey the essence of the language design. Some motivation for the design of language features is included but are necessarily terse. For a more complete description, see the "ICC++ Language Reference Manual."

1 Concurrency

To introduce concurrency, the programmer relaxes the sequential order of a program, specifying a partial order between statements. In ICC++, partial orders are specified using `conc` blocks.[2] The definition of `conc` preserves a sequential view of local variables, and expresses concurrency with respect to shared objects. Syntax:

$$\texttt{conc } \{ \ S_1 \texttt{;} \ \ldots \ \ \texttt{;} \ S_n \texttt{;} \ \}$$

Defines a partial order $\prec$ on the statements:

$$S_i \prec S_j \Longleftrightarrow i < j \ and \ S_i \Rightarrow S_j$$

where $S_i \Rightarrow S_j$ indicates that statement S_j is dependent on S_i. Statements in a `conc` block are executed such that if $S_i \Rightarrow S_j$, then S_i will be executed completely before S_j is begun. S_j is dependent on S_i if

1. any identifier which appears in both S_i and S_j is assigned in one of them,

[1] The authors gratefully acknowledge the contribution of many others to the language design, notably John Plevyak, Vijay Karamcheti, and Julian Dolby. Others who have been involved include Professors Sam Kamin, Sanjay Kale, and Prith Banerjee. In addition, Rob Hasker, Howard Huang, Steven Parkes, and T.K. Lakshman have contributed to the design effort. Any flaws in this document or language design are solely the responsibility of the authors.

[2] Alternatively the `conc` declarations can be viewed as pragmas on the blocks, directing the compiler to allow these statements to be reordered or executed in parallel subject to the following rules.

2. if S_i contains a jump statement.

The first rule ensures that basic data dependences on local variables are enforced. It also allows object operations (via pointers or refs) such as `a->b(); a->c();` to execute concurrently. How this concurrency is controlled is discussed in Section 2. Jump statements (as defined in *The Annotated C++ Reference Manual*) in the second rule gives traditional C++ control flow operations such as `continue` and `break` natural semantics, serializing the `conc` block if they are executed. A `conc` block exits after all statements within it have returned. As in C++, nested blocks are treated as statements.

2 Objects and Concurrency

While the introduction of parallelism is a control structure issue, control of concurrency is a critical issue for encapsulation. In order to support sharing patterns required by dynamic and reference based data structures, ICC++ extends C++ objects with implicit concurrency control. Concurrent objects in ICC++ are defined and created in a fashion similar to C++:

```
class Set {
  int member_count;
  integral Element my_elements[];
  void insert(Thing);
};

Set aSet;
```

where `Element` is a previously-defined class. The code above defines a class with an `int` field, an `Element` array field, and an `insert` operation, then creates an instance of the class, `aSet`. The `integral` annotation indicates that `my_elements[]` should be considered part of the object's state for the purposes of concurrency control (see below). Member variables are read and written via accessor functions (defined automatically), so member variable access also fall under this concurrency control. For example, for an instance variable `a` of type `B`, there are accessor functions `a(void)` and `void operator.a=(B)`. Declaring member variables as `private` makes these accessor functions private. At present accessors cannot be overridden, but this may change in future language revisions. ICC++ calls constructors as in C++, but destructors are more complex. For auto variables, destructors are called when the variable goes out of scope. But, because ICC++ provides automatic storage management, destructors for all other objects are called only when an object is garbage collected (i.e. delete may not cause the destructor to be called immediately).

2.1 Concurrency Control

ICC++ constrains the execution of method invocations on an object so that intermediate states created within a member function are not visible. Such method invocations are executed concurrently but in such a way that the effect on the member variables is as if the methods operated one after another. The `integral` declaration extends this notion of consistency to include the state declared `integral`. For example, if two `insert` operations on a `Set` updated the `my_elements[]` array, those updates would not be interleaved. Syntactively recursive calls (i.e. calls on `this`) to member functions are allowed and may expose intermediate states. However, such situations are routinely managed by programmers in sequential programs, so no additional language support is provided. `friend` functions are considered as members of all the objects for which they are friends,

and thus can be used to procedurally compose operations on several objects into a single consistent operation. In a distributed memory setting, such multi-object operations can be expensive, so they should be used carefully.

2.2 Concurrency Guarantees

In a concurrent language, a programmer must be able to reason about concurrency to ensure progress. ICC++ guarantees that all member function invocations for which the order of execution explicitly cannot affect final object state (determined by trivial examination of the methods) are guaranteed to execute in parallel.[3] Specific examples for which concurrency is guaranteed include two methods which share no member variables and those that employ read-only sharing of member variables. For syntactically recursive calls, order independence is based on textual inclusion of recursively called methods. Note that these concurrency control rules apply to uses of objects pointed to by `integral` member variables as well.

3 Collections

Collections are an important organizing structure for parallel programs because they form a convenient and natural basis for expression of parallelism as well as the distribution of data. ICC++ extends C++ arrays to produce collections. Collections are objects that encapsulate a set of elements (collection elements), and additional collection state. This integration enables arrays to be manipulated as objects (allowing collection member functions, for example). Because elements are aware that they are part of a collection, they can help implement its composite behavior. In ICC++, object collections are defined as are standard classes, but with a `[]` appended to the end of a class name.[4]

Member variables and member functions can be defined for the element or collection. Collection members are explicitly qualified by the collection type. Each collection element has a private set of the element members, and the collection members form a separate object which is shared across all elements. For example:

```
class Counter[] {
    int count;
    int Counter[]::total;

    int thecount(void) { return count; }
    int Counter[]::thetotal(void) { return total; }
};

Counter mycounter[10];
```

Defines a collection type `Counter[]` and an element type `Counter`, exposing distinct names for the element and collection types. The elements each have one member variable `count` and there is one collection member variable `total`. There are also two member functions: a collection member function, `thetotal()`, and an element member function, `thecount()`. Notice that `count`, an element member variable is scoped within element member function `thecount()`, and `total` is

[3] Write races that deposit the same value are a tricky case, but ICC++ does not guarantee concurrency for such cases.

[4] For compatibility, one can still define array-style collections implicitly. However, if additional collection member functions or variables are required, an explicit declaration must be used.

scoped with the collection member function `thetotal()`. However, element member variables are not available in collection member functions, nor are collection member variables available within element member functions. The natural consequence of this scoping rule and the serialization model for objects is that element member functions can execute on distinct elements concurrently, and furthermore, element and collection member functions can also execute concurrently.

3.1 Predefined Member Functions

The member functions below are defined on all collections and their elements. For nested collections (collections of collections), both sets of methods are defined for the types other than the outermost collection and innermost element.

Collection member functions:

`[ ]`	The subscripting operator indexes elements of the collection. When applied with no argument, an arbitrary element in the collection is returned.
`size`	Returns the number of elements in the collection.
`nearest`	Returns the nearest element in the collection

Element member functions:

`elementtype[ ]::this`	In an object of type `elementtype`, returns a pointer to the enclosing collection.
`index`	Returns the element's index within the collection.

3.2 Derivation

Collections can inherit from standard classes as well as from other collections. Nested collections can inherit from nested collections of equal or lesser order. For example, a collection can inherit from an ordinary C++ class. Then, the element type is a subclass of the standard class. As shown in Figure 1, the `DistAccumulator[]` collection is derived from the `Accumulator` class, and the `DistAverage[]` collection is derived from the `DistAccumulator[]` collection type.

3.3 Templates

Templates can be exploited to reuse collection structure, much as they are used with ordinary classes. The example below defines a numerical collection with a range of arithmetic functions. The last two lines are collection declarations, and the lattermost assumes that a `complex` class has been defined elsewhere. Of course, more complex collections can also be defined.

```
template <class T> class Numerical[] : public T {
        Numerical[]::global_sum();
        Numerical[]::min();
};

Numerical<float> X[50];
Numerical<complex> Y[50];
```

3.4 Compatibility with Arrays

Collections in ICC++ support all of the functionality of built-in C++ arrays. They can be implicitly or explicitly defined, indexed, and passed around. As discussed in Section 4, they cannot be

```
class Accumulator {
  int total;
  int sum(void) { return total; }
  int accumulate(int i) { return (total += i);}
};

class DistAccumulator[] : public Accumulator {
  int DistAccumulator[]::sum(void) {
       int subtotal = 0;
       for(int i = 0; i < size; i++)
          subtotal += (*this)[i].sum();
       return subtotal;
  }
  int DistAccumulator[]::accumulate(int i) {
      return (*this)[].accumulate(i);             // into an arbitrary element
  }
};

class DistAverage[] : public DistAccumulator[] {
  int count;

  int collect(int i) {
    accumulate(i);              // Element method, accumulates value
    count++;                    // count the nr of items
  }
  int DistAverage[]::average(void) {
    int count = 0;
    conc for(int i = 0; i < size; i++)
      count += (*this)[i].count;
    return sum()/count;
  }
  int DistAverage[]::collect(int i) {
    (*this)[].collect(i);
  }
};
```

Figure 1: Derivation with Collections

represented by pointers (must be declared as collections), and consequently, pointer arithmetic is not a meaningful operation, array subscripting must be used instead.

3.5 Concurrent Loops

Each of the C++ looping constructs can be modified by `conc`: `conc for`, `conc while`, and `conc do while`. Because looping constructs form a natural basis for expressing parallelism, ICC++ employs a simple semantics which exposes cross-iteration concurrency.

```
conc for(a;b;c){
       d;
     }

conc while (b){
        d;
        c;}

conc do {d;
         c;}
     while b;
```

Figure 2: Concurrent looping constructs.

For each cases, adding `conc` indicates a parallel loop. Loop carried dependences are respected for scalar variables; others – array dependences and those through pointer structures – must be enforced explicitly by the programmer. Generally, loops with such dependences will be written as serial loops. Subject to the minimal serialization required to enforce the scalar inter-iteration dependences, the loop iterations may all execute concurrently. However, there are no concurrency guarantees, a sequential execution must be acceptable. As in sequential loops in C++, `continue` and `break` skip the remainder of the iteration in which they are executed and exit the loop respectively. This feature allows parallel loops with more complex control flow to be parallelized optimistically.

4 Restrictions for Program Optimization

ICC++ supports much of C++, but with several restrictions to enable program analysis, optimization, and automatic storage management. These restrictions have the affect of hiding the underlying storage model from the programmer, and are similar to those proposed by Ellis and Detlefs, but are less restrictive.[5] The following C++ features are disallowed:

1. Interconverting arrays and pointers
2. Pointers to simple types (int, float, char, etc.), with the exception of those needed for interfacing to external C libraries
3. Unsafe casts
4. Union types

[5] Ellis and Detlefs, *Safe Efficient Garbage Collection for C++*, DEC Systems Research Center Technical Report 102, July 1993.

Note that C++ forbids pointer arithmetic on pointers, except within the confines of an array, and ICC++ prohibits the use of pointers to refer to arrays, therefore pointer arithmetic is effectively disallowed. Union types should be represented via derivation.

ICC++ allows casts that it can check at compile or runtime, but forbids those that are reinterpretations of memory. In terms of the new cast syntax adopted in the C++ draft standard, this means that `static_cast<T>` is allowed, but some of the casts will be checked at runtime and hence may generate runtime errors. `dynamic_cast<T*>` is permitted and checked at runtime. const_cast<T> is also permitted, but reinterpret_cast<T> is excluded.[6]

5 Advanced Concurrent Programming Features

`spawn` *s*;

creates a new thread, executing the statement *s*, and the spawn statement completes immediately. *s* can be any ICC++ statement. Identifiers of simple type become read-only in *s*, preventing unsynchronized interaction between the spawning and the spawned threads. The spawned and spawning threads are scheduled fairly.

`return` *s*;

The `return` statement is similar to that in C++ returning a value to the current function's caller and resuming execution of that caller. If there is no return, the compiler will automatically insert one if the procedure function does not use `reply` in any way.

`void reply(`τ`);` // τ is the return type of the function

ICC++ provides a `reply` function with prototype shown above for each user-defined function. `reply` returns a value to the function's caller, but does not terminate execution of the current function, allowing caller/callee concurrency. Since `reply` is a function, a pointer to it can be captured and passed out of the function.

Reply functions can be used to delegate responsibility for producing an answer. For example, to support an explicit continuation passing style, programmers can use `void` functions for each such continuation passing call, and `spawn` to generate concurrency. For example, `spawn reply(forward_call(...))` implements tail forwarding where `forward_call(...)` executes and its return value is used as the return value of the current function.

6 Performance Pragmas

ICC++ also provides a variety of performance pragmas which allow the programmer to communicate information to the compiler and to control the behavior of the runtime system. Examples include `map-aggregate`, and `new-local` which provide placement control for collections and individual objects respectively. `local(a,b)` tests the relative position of two objects. Additional performance pragmas are being explored for object caching and consistency, as well as for controlling compiler speculation.

[6] See *The Design and Evolution of C++* for a detailed discussion of the new cast syntax.