Compositional C++: Compositional Parallel Programming

K. Mani Chandy Carl Kesselman
California Institute of Technology

Keywords: Parallel processing, Object-oriented programming, C++, Compositional programming.

Abstract

A compositional parallel program is a program constructed by composing component programs in parallel, where the composed program inherits properties of its components. In this paper, we describe a *small* extension of C++ called Compositional C++ or CC++ which is an object-oriented notation that supports compositional parallel programming. CC++ integrates different paradigms of parallel programming: data-parallel, task-parallel and object-parallel paradigms; imperative and declarative programming; shared memory and message-based programs. CC++ is designed to be transportable across a range of MIMD architectures.

1 Introduction

1.1 Compositionality

Programming languages provide mechanisms for structuring programs as a composition of component units. Properties of programs composed using sequential or functional composition can be derived from properties of their components. For example,

$$wp(F; G, Q) = wp(F, wp(G, Q))$$

where wp denotes weakest precondition, and ";" denotes sequential composition, and Q is the desired postcondition of $\{F;G\}$. Functional composition, as in $f(g(x))$, is defined in mathematics. Sequential composition and functional composition follow the principle of implementation hiding: programmers can compose program specifications without being concerned with program implementation.

In parallel programs, however, implementation hiding is unsatisfactory. To prove properties of a program composed in parallel we must (in general) give proofs for the components such that one component does not interfere with the proof for the other. To prove properties of the composed program we have to use details of the components which we would rather hide.

One way of hiding implementation is to design parallel composition such that properties of components are also properties of the composed program. In this case we need not concern ourselves with implementations of the components, nor need we be concerned about one component interfering with the proof of another. The key issue, then, is to design parallel composition so that the whole inherits properties of the parts. We introduce a form of parallel composition, called *proper* parallel composition, that has this *inheritance property*. We begin our discussion of proper parallel composition with the question: what does "property p holds in program F" mean?

In *closed-system* specifications, p holds in F means that p holds in all computations of F [23]. Here, computations of F refer to computations of F executing by itself — the environment of F does not execute. With closed-system specifications it is possible that a property holds both for F, and for G, but not for F composed in parallel with G. In general, closed-system specifications do not compose. See UNITY [11] for a detailed discussion of this issue.

In *open-system* specifications, "p holds in F" means that p holds for all computations of F composed in parallel with G, for *any* program G. Open-system specifications enjoy the inheritance property: the composed program inherits properties of its components. Unfortunately, properties in open-system specifications can be too weak to be useful because a property of a program F must hold when F is composed in parallel with *any* program, including programs that the designer of F had not intended to be composed with F.

Proper composition takes the open-system approach, but only programs that obey certain rules about naming and assignment can be composed using

proper parallel composition [21, 12]. The rules for proper parallel composition are designed to yield the inheritance property. The central idea in obtaining compositionality is that a process has private variables that cannot be referenced by other processes, and shared variables that can be assigned values at most once. (A message sequence can be thought of as a list in which each element — a message — is assigned a value when the message is sent.) Examples of the use of single assignment variables can be found in [1, 15, 20].

1.2 Programming Languages for Compositional Programming

We turn now to the question of what programming notation should be used to express proper parallel composition in a program. The advantages and disadvantages of introducing a new parallel programming language are well known [18]. A significant advantage of using small extensions to widely-used sequential languages is that programmers can learn the extensions quickly, and use tools developed for the base language. This is the approach we have taken, using C++ [13] as our base language. We choose C++ because of its widespread use, and its support for abstract data types, object-oriented programming, and the specification of generic algorithms through templates and inheritance.

In the remainder of this paper, we show how C++ can be augmented to create a parallel programming language. We call the resulting language Compositional C++ or CC++. The following discussion will focus on the essential aspects of CC++; a complete description of CC++ can be found in [8, 10]. CC++ is a flexible, concurrent object-oriented programming language particularly well suited to the implementation of large-scale parallel programs and high performance parallel programming libraries utilizing a range of parallel programming paradigms.

Prior to a discussion of CC++ specifics, we make a few observations that have driven the design of CC++.

Determinism and Nondeterminism Programmers developing scientific applications want their programs to be deterministic: executions of the same program with the same inputs should produce the same outputs. Numerical analysts craft careful sequences of steps to avoid numerical instability, and

they need to be guaranteed that the same sequence of steps will be executed in each execution. By contrast, reactive systems are nondeterministic because they deal with an uncertain environment.

Programmers can guarantee that CC++ programs are deterministic by following certain simple conventions — if they follow these conventions, then they have no proof obligation in demonstrating determinism. Programmers can, if they wish, use nondeterministic constructs. The ability to choose between deterministic and nondeterministic constructs, and to compose deterministic and nondeterministic programs, allows programmers to develop a variety of applications including reactive systems with components that are scientific applications.

Paradigm Integration No "best" parallel programming paradigm exists. Semaphores, monitors, message passing, etc. all have their uses. Different paradigms can be appropriate even with a single program. CC++ was designed to facilitate the use of a different programming paradigms, such as:

- **Task, Data, and Object Parallel**. Distributed arrays and distributed grids can be defined as classes in CC++, allowing data-parallel computations on these objects. In addition, CC++ supports cooperative processing and task parallelism. Furthermore, since CC++ is object-oriented, it supports the object-oriented paradigm.

- **Declarative and Imperative**. The extensions to C++ allow declarative programs to be written in CC++. Programmers can, however, continue to use the familiar imperative style of C++.

- **Shared Variables and Message-Passing**. Though processes in CC++ share variables, libraries of message-passing channels are provided so that the message-passing paradigm can be used if desired. Libraries of semaphores and monitors allow programmers to use the styles of their choice.

2 Compositional C++

As its name implies, CC++ is based on C++ [24]. C++ is itself is an extension of the **C** programming language. C++ adds strict typing, function

overloading, encapsulation, abstract data types and object-oriented programming to **C**. These features make C++ a good language for implementing program libraries and large scale software systems.

A central concept in C++ is the *class*, which combines code and data into a single unit, or object. The data components of a class are called *data members*. The functions associated with a class are called *member functions*; in other object-oriented languages, member functions are often called *methods*. The subset of the data members and member functions that define the interface to the class are declared *public* and are accessible from outside the class. All other members are private and they are accessible only to member functions of the class. A class, therefore, forms a unit of encapsulation.

A member function can only be called from an object, or a reference to an object. For example, if `Obj` is a variable whose type is a class containing a member function `f`, `f` is invoked from `Obj` by the expression: `Obj.f()`. If `ObjPtr` is a pointer to `Obj`, then `ObjPtr->f()` invokes `f` as well.

The advantages of an object-oriented approach to the design of parallel systems has long been recognized [17]. Object-oriented systems provide well-defined interfaces, co-location of function and data, encapsulation of data and data abstraction. These features encourage the construction of scalable parallel systems. Consequently, the encapsulation and object-oriented features of C++ have made it the basis of a number of parallel programming systems [16, 5, 25]. However, we will show that our points of departure — declarative and compositional programming — result in a significantly different approach to parallel programming in C++.

CC++ is a pure superset of C++; it consists of C++ plus seven extensions. These extensions impact the language in the following areas:

- flow of control,
- synchronization and communication,
- nondeterminism,
- locality and heterogeneity.

With the exception of the extensions supporting locality and heterogeneity, the C++ constructs extended by CC++ are part of ANSI C as well. Thus

C programmers can benefit from CC++. In extending C++ to CC++, the new constructs were carefully designed to be a minimal and complete set and to support formal methods. Proving properties of CC++ programs is beyond the scope of this paper, but is discussed more fully in [9, 6].

2.1 Flow of Control in CC++

CC++ is intended to be a general purpose parallel programming language. As such, parallelism in a CC++ program is explicit. Three constructs are available for parallel composition in CC++: the `par` block, the `parfor` statement and the `spawn` statement. We shall start the discussion with a description of `par` blocks.

The `par` Block. The `par` block is the most basic means of specifying parallel composition in CC++. Its syntax is that of the compound statement in C++ with the keyword `par` preceding the block. An example of a `par` block is found in Figure 1. A `par` block can lexically contain any CC++ statement with the exception of the `return` statement and variable declarations.[1] As seen in Figure 1, the statements in the block can be sequential blocks and `par` blocks can be nested.

```
par {
  procedure1();
  { procedure2();
    par { procedure3(); procedure4(); }
    procedure5();
  }
  procedure6();
}
```

Figure 1: An example of a `par` block

A new thread of control is created for each top level statement in a `par` block. A `par` block terminates when all its statements terminate. However,

[1] A `goto` statement is allowed in a `par` block, but its use is restricted.

there is no requirement that a `par` block terminate. Parallel execution of the statements in the `par` block is defined by fair, interleaved execution of the top level statements in the block. A definition of fairness is given in Section 2.2; however, for the time being, take fairness to mean that every executable statement in a `par` block will execute eventually, even if the `par` block does not terminate.

With the exception of atomic functions, which are discussed in Section 2.3, the granularity at which the interleaving occurs is not defined. As an example, consider the `par` of Figure 2. Assume that all the statements in the `par` block terminate. Possible execution orderings of the `par` block include (but are not limited to):

a1 a2 a3 b1 b2 b3
a1 b1 b2 b3 a2 a3

The sequential ordering of the function calls within statements $S1$ and $S2$ is maintained. However, even though these statements are sequential, their execution is interleaved. Note that execution of a_i and b_j can be interleaved too.

```
par {
  { a1(); a2(); a3(); } // Statement S1
  { b1(); b2(); b3(); } // Statement S2
}
```

Figure 2: Parallel execution of two sequential blocks

The ability to create a collection of threads is not too useful unless threads can interact with each other. Communication and synchronization between threads is discussed in Section 2.2. We only note here that the primary communication mechanism in CC++ is shared variables. This is not to say, however, that CC++ can only execute efficiently on a shared-memory computer. As we will see in Section 2.2, the sharing is constrained in such a way as to make efficient execution on a range of parallel architectures possible.

The `parfor` Statement. A `par` is useful when one needs to create a fixed number of threads and that number is known at compile time. While re-

cursion within a `par` block can be used to create an arbitrary number of concurrent threads, iteration is often a more natural and convenient means of expressing such computations. For this reason, CC++ has a parallel loop construct, the `parfor`.

The syntax of a `parfor` statement is exactly like the `for` statement in C++. The statement specifies loop initialization, a termination test, an index-update expression and the loop body. The loop body can be a simple statement, a sequential block, or a parblock. Note that in C++, the index variable of a loop is not constrained to be an integer and the termination test and index update expression can be any valid C++ expression. An example of a `parfor` statement is shown in Figure 3. Notice the use of the C++ feature that allows the declaration of an index variable to be placed in the `parfor` statement itself.

```
parfor (int index = 0 ; index < N ; index++) {
  a1(index); a2(index+1); a3(index);
}
```

Figure 3: A `parfor` statement

Each iteration of a `parfor` creates a new thread which executes in parallel with all other iteration bodies (and the rest of the computation as well). The threads have the same interleaved execution semantics of the `par` block. When all the loop bodies terminate, the `parfor` statement terminates. Variables declared in the initialization part of a `parfor` loop receive special treatment. Within each thread, a local copy of the each index variable is created and initialized with the values of the index variables at the time of thread creation. Thus in Figure 3, each loop body will have a local variable called `index` and its value will be set to the value of `index` in the `parfor` loop at the time the thread for the loop body was created. Regardless of the execution order of the loop bodies, the correct value of `index` will be available in the loop body.

The `spawn` Statement. The third and final method for specifying concurrent execution in CC++ is the `spawn` statement. The termination criteria associated with `par` blocks and `parfor` statements imposes structure on a

concurrent computation. When the statement terminates, all parallel computation is complete and any postconditions associated with the block hold. However, there are situations in which this structure is a hindrance rather than a help. An example is a program that sets up a network of interconnected servers. In this case, we would like to start a new thread of execution for each server and have the program proceed, independent of the termination of the thread. While this could be done using a `parfor` or `par` block, one has to go to some effort to work around the fact that the servers must terminate before the statement after the `par` block or `parfor` statement executions. The situation would be simplified if one could start a new thread of control and proceed to the next statement immediately. This is exactly what the `spawn` statement does. An example of a `spawn` statement is found in Figure 4.

A `spawn` statement executes an arbitrary CC++ expression in a new thread of control. The execution of the statement is interleaved fairly with the rest of the program. Unlike the `par` and `parfor`, a `spawn` statement terminates immediately, regardless of the status of the process that was spawned.

```
spawn x + y + g(z);
```

Figure 4: A `spawn` statement

Comparison With Other Approaches. Parallel composition as defined by `par` blocks can be found in a number of other parallel programming notations. For example, a `par` is equivalent to the use of *cobegin* and *coend* in [22] and the parallel composition operator in PCN [12].

The use of `par` blocks differs from most other concurrent object-oriented languages in that with a `par` block, multiple threads of control exist within a single object. Other languages tend to associate thread creation with object creation [5, 16, 25, 2]. Consequently, only one thread of control is ever associated with an object. The `spawn` statement in CC++ can be used to achieve the same effect.

There are two advantages to `par` (and `parfor`) over a one-to-one correspondence between objects and threads. First, these statements are block-oriented, and they make parallelism within a block explicit; there is no ques-

tion as to which statements execute in parallel and which in sequence. The second advantage is that one can associate a post condition with a `par` block or a `parfor` statement. When the statement terminates, the post condition can be asserted. This simplifies the process of reasoning about the behavior of the program.

Reasoning about the behavior of a program containing a `spawn` statement is more difficult. Because there is no way of knowing when the thread started by the `spawn` starts or completes, assertions about the `spawn` statement must state that a condition will hold at some unknown point in the future. Thus one has no choice but to resort to a temporal operator such as the *leads-to* operator [11].

2.2 Synchronization and Communication

In the previous section, we saw how concurrent computations can be created by the use of `par`, `parfor`, and `spawn` statements. In this section, we will discuss how concurrently executing program components can share information and synchronize.

Before we discuss CC++, a brief review of type modifiers in C++ is in order. In C++, as well as in ANSI **C**, the type of a variable can be modified by the keyword `const`. The object named by a `const` variable has a constant value and cannot be modified. [2] Figure 5 illustrates a number of `const` object declarations and explains their meaning. For obvious reasons, a constant object must be initialized at the time the object is created.

From the point of view of concurrency, `const` objects are very useful. Once a `const` object is created, its value is known for the entire lifetime of the object. Since threads are not allowed to write to a `const` object, there are no race conditions associated with it; threads can read the value of a `const` object without restriction. From an implementation point of view, `const` objects are useful because they can be copied into more than one processor without concern for maintaining consistency between the copies.

The requirement that a `const` object be initialized at creation time is too restrictive to make constants the only basis for communication in a parallel

[2] Under certain circumstances, the C++ type system can be manipulated in such a way as to allow the contents of a `const` variable to be altered. This may or may not result in a runtime exception. We will not consider such operations in the present discussion.

```
// x is a regular integer.
int x;

// const_x is a constant integer, initialized to 23.
const int const_x = 23;

// y is a constant pointer to a constant integer.
// Neither the pointer nor the integer can be altered.
// y is initialized to point to const_x
const int * const y = &const_x;

// z is a constant pointer to a integer.
// The pointer cannot be changed, though the contents of the integer can.
// z is initialized to point to x.
int * const z = & x;

// w is a pointer to a constant integer.
// The object being pointed to cannot be changed,
// but different objects can be pointed to.
// w does not have to be initialized.
const int * w;
```

Figure 5: Examples of the use of the `const` type modifier in C++

program. However, the concept can be extended to form the basis of a flexible communication structure. To do so, CC++ has an additional type modifier: `sync`. A `sync` modifier indicates that a variable is used for synchronization.

A `sync` variable in CC++ is treated *exactly* like a `const` variable in plain C++ with two exceptions:

- Unlike a `const` object, the initial value of a `sync` object does not need to be provided at creation time. Initialization can be delayed. A `sync` object is initialized by assigning it a value. Once initialized, a `sync` object cannot be assigned to again. It is an error to reinitialize a sync object.

- Any attempt to read the value of an uninitialized `sync` object is delayed until some finite time after the `sync` object is initialized.

Subject to the restrictions on reinitialization, a `sync` object can be used wherever a regular object can be used. For example, data members of a class can be all `sync`, all non-`sync` or some combination of the two. Structures can be initialized incrementally; it is only when accessing a noninitialized component of a structure that the execution of a thread is blocked.

The `sync` variable provides a single mechanism for synchronization and communication between concurrently executing threads. For communication between threads, two or more threads share a common `sync` object and one thread writes a value into the object. Synchronization is achieved in that the thread reading the value cannot proceed until the value has been written. Notice that the degree of sharing is not limited; any number of threads can reference a single `sync` object.

From an implementation point of view, `sync` objects have many of the same properties as `const` object; copies of `sync` objects can be cached on any number of processors. While copies of a `sync` object can be inconsistent, a copy can be inconsistent in only one way. A copy of an object can be uninitialized on some processors and initialized on others. The copies that are initialized will all contain the same value. In the situation where a local copy of a `sync` variable is out of date, any attempt to read its value will block and the correct value of the variable will be obtained. These semantics can be efficiently implemented on both shared memory and distributed memory parallel computers; hence, `sync` variables provides a unified, architecture-independent means of synchronization and communication.

In CC++, all communication takes place via variables shared between concurrently executing threads. A parallel composition is *proper* if all the variables shared by the procedures being composed are `sync` variables. Proper parallel composition enjoys the inheritance property: properties of components are also properties of the composed program.

Comparison with Other Approaches No doubt, some readers will have recognized sync variables as being single assignment variables from dataflow languages [20] or from languages based in concurrent logic programming [15]. A reference to a structure which contains `sync` objects behaves much like an I-Structure in the dataflow language ID [3]. `sync` variables differ in that the `sync` attribute can be extended to abstract and concrete data types as definable in C++. In addition, `sync` variables differ from variables in programming languages such as Strand [15] and PCN [12] in that the blocking rule for `sync` variables prohibits the use of variable-to-variable assignment found in these languages.

Many concurrent object-oriented languages [2, 4, 26] use function call as the basis for communication. In actor-based languages, a function call is interpreted as sending a message to the target object to perform the requested operation. The arguments in the function call are passed to the target object as well. The function call terminates immediately, without a waiting for a response from the target object. Applying this approach to C++ is problematic in that this approach changes the meaning of function call. By associating communication with shared variables and assignment, the semantics of all of the underlying operations in C++ are preserved. Finally, we note that actor type semantics of function call can be achieved either through the use of the `spawn` statement within the body of a member function or assignment to a `sync` variable whose value is read by a nondeterministic fair merger.

Fairness. With the introduction of `sync` variables, we can now define fairness. At any point in a computation, a thread can be in one of two states: **executable** or **suspended**. A thread is executable when it is created; a thread becomes suspended when it attempts to read an uninitialized `sync` variable. In CC++, an executable thread remains executable until it is executed; the execution of some other threads cannot make an executable process become suspended.

The fairness rule implemented by CC++ is as follows. At all points `t` in the computation, the following holds:

1. The program terminates (within a finite number of steps from `t`), or
2. for every thread `r` that is executable at point `t`: the computation of `r` will progress (within a finite number of steps from `t`).

2.3 Nondeterminism

In this section, we address nondeterminism in CC++ programs. If all parallel compositions are proper, the resulting program is guaranteed to be deterministic if there are no runtime errors. Deterministic behavior simplifies program debugging, testing and error analysis. In CC++, nondeterministic behavior is produced when concurrent threads share a non-`sync` variable and at least one thread updates the contents of that variable. Modifications to such a variable are unordered and any thread reading the variable can have nondeterministic behavior. Note that the sharing does not have to be direct; the variable to be modified can be encapsulated in an object and the update made by a member function of that object.

Some control over how modifications to shared variables take place is required. Reads and writes of built-in data types (ints, floats, pointers, but not user defined objects) are atomic. Also, CC++ has the concept of a *atomic function*. The semantics of an atomic function are that the execution of an atomic function will not be interleaved with any other statement in a computation. An atomic function is used to establish the minimum granularity at which interleaving can take place in the execution model.

A CC++ function that satisfies the constraints described later can be declared to be atomic. However, atomic functions are most useful when they are members of a C++ class. Atomic functions are not allowed to access `sync` variables, nor can they access variables outside of an object for which an atomic function is a member. These restrictions allow the semantics to be fulfilled by ensuring that only one atomic function per class instance executes at a time.

2.4 Data Locality and Heterogeneity

Data locality is an important aspect of both sequential and parallel programs. Locality determines how effectively a program utilizes the memory hierarchy of a computer system. While it is certainly in the purview of the compiler to preserve and enhance the data locality present in a program, it is ultimately the responsibility of the programmer to specify a program with suitable data locality.

In sequential programs, locality of reference is achieved by carefully positioning data elements in data structures, tuning algorithms to exploit characteristics of the memory hierarchy of a specific computer and introducing explicit structures, such as software buffers and caches. While such devices are available to parallel programs as well, use of these techniques is complicated due to the increased complexity found in parallel programs. To facilitate the construction of algorithms that can exploit locality in CC++, we provide an additional mechanism for grouping objects together. This CC++ construct is called a *logical processor object*. In addition to providing a way of specifying locality, processor objects also form the basic mechanism for constructing heterogeneous software systems.

A C++ program consists of a set declarations. A declaration can declare a variable, define a function, class or data type. Each program contains some number of declarations that are known to the whole program, that is they are global.

CC++ deviates from C++ in that all global declarations are contained in a special class called the *logical processor* class. The logical processor class is not defined in the program text. Rather, it is defined by the user externally and is created by a system building utility, such as the linker. Functions that would be global in C++ become public member functions of the logical processor class for the program. Likewise, class declarations and type declarations become nested declarations in the processor class.

There can be more than one type of processor class in a CC++ computation. Each logical processor object is a complete, independent CC++ program, except that one logical processor object can refer to another logical processor object via a pointer to that logical processor, or via a global pointer, which is described later. In C++ terminology, we say that all members of a processor class have internal linkage. In particular, variables declared static are not shared between instances of processor objects.

Because all linkage is within a single logical processor object, there is no name space outside of logical processor objects. Global members of a processor object can only be accessed via a pointer to that processor object. An important ramification of processor objects is that all object creation within a processor object is done using a local version of the `new` operator. Thus both dynamically and statically allocated objects are local to a specific processor object.

The operators that access a logical processor can be overloaded, eliminating the need for an application program to ever refer directly to a processor object. This provides a flexible, user-defined scheme for addressing logical processors (for examples, see [14]). Examples of how a user might view logical processor ids include an integer, a pair of integers (if the logical processors form a two-dimensional mesh), or an enumerated type.

When a computation starts, it has a single processor object mapped onto a single processor. Additional processor objects can be dynamically created by the `new` operator.[3] Processor objects can be mapped onto a specific computational resource through use of the placement argument to the `new` function. For example, the statement:

```
proc_t * sPtr = new (23) Solver;
```

creates a new instance of the program *Solver* and maps that instance onto node 23. The placement argument can be of any type allowing for sophisticated mapping strategies; if no placement argument is specified, the new processor object is co-located with the current processor object.

Global Variables. A major consideration in the design of CC++ was to be a complete superset of C++. One problematic aspect of this requirement is that CC++ must support global variables, file scope variables and static variables. Logical processor objects provide a good mechanism for handling these types of variables.

Recall that global declarations in a program become public members of the logical processor object constructed for that program. Also, each instance of a processor object is independent. Consequently, a global or static variable

[3]The `new` operator is used to dynamically allocate memory of for an object of a specific type in C++. A `new` not only allocated the needed memory but also ensures that the memory is properly initialized according to the type of the object requested.

is local to a processor object. If more than one instance of a processor object exists, then each one has its own "local global". If a function in the processor object refers to a global variable, the object used is located in the processor object in which the function is located.

It can be argued that our treatment of global variables has the disadvantage that programmers versed in C++ semantics would expect a global to be global to an entire computation. Furthermore, a smoother transition from a sequential program to a parallel program would be obtained if a global name were global to a computation rather than global to the processor object. We counter this argument by observing that in the degenerate case of a single processor object, CC++ globals behave exactly like C++ globals. Computations with more than one processor object are no longer C++ programs and some deviation from C++ behavior should not be unexpected. Furthermore, a parallel language should not produce by default a construct as completely unscalable as a globally shared variable.

There are several advantages to our approach. Because it is a natural consequence of processor objects we do not need to introduce a new concept to explain global variables. This minimizes the number of concepts that a programmer must understand in order to use CC++. In addition, a frequent use of global variables is to reference large data structures which are subject to domain decomposition when parallelizing a program. In these situations, a "local global" is the right concept. Finally we note that global distributed objects can be built from processor object "local globals" [19].

Global Pointers. In C++, a reference to an object can be stored in a pointer variable or a reference variable. In CC++, pointers and references can be used as well. However, in CC++ we must distinguish between the situations in which a pointer or reference variable resides in the same logical processor as the object being referenced or a different logical processor from the object being referenced. The first case is referred to as a *local* pointer or reference; the second case is referred to as a *global* pointer or reference. Thus in CC++, a pointer can be either a local pointer or a global pointer. For clarity, the following discussion will be limited to pointers; however, it applies to references as well.

In Figure 5 we saw that C++ allows a pointer to be modified by the keyword `const`. In CC++, the modifiers that can be applied to a pointer

are augmented to include the keywords `global` and `sync`. A pointer that is not declared to be a global pointer is a local pointer. Thus, the statement:

```
int * global g_ptr;
```

declares `g_ptr` to be a global pointer to an integer. More than one modifier can be used in a declaration, thus the statement:

```
int * global const g_ptr;
```

declares a variable that is a constant, global pointer (i.e. an inter-logical processor pointer whose value cannot be changed).

A pointer in one logical processor object that points to a location in another processor object must be a global pointer. A pointer in one logical processor object that points to a location within the same logical processor object can be either a local pointer or a global pointer. If a `global` pointer points to a `sync` object, the CC++ implementation can cache the contents of the pointer in the processor object containing the pointer. Although the correct semantics are maintained regardless of the pointer type, efficient program execution on a wide range of architectures favors the use of global pointers to `sync` objects (when possible) over global pointers to non-`sync` objects.

Global pointers provide a mechanism for remote procedure call. If one holds a global pointer to an object, then a member function in that object can be invoked through the global pointer. The member function executes in the processor object that contains the target object, not in the processor object that contains the pointer.

This concludes our discussion of CC++. In the next section, we will demonstrate how CC++ is used in a small example.

3 A CC++ Programming Example

In this section, we illustrate how CC++ is used to write a parallel program with a small example: a queueing simulation. A central point about this program is that the CC++ program is identical to a C++ program for the same problem, except that some variables are declared to be `sync` and a `for` loop is replaced by a `parfor` loop.

The structure of the program is shown in Figure 6. The simulation consists of a number of queue elements. The input to a queue element is a sequence of the times at which service requests are made and a sequence of service requirements in terms of seconds. The output of a queue element is the sequence of times at which service requests complete. Interarrival times for the first queue element are generated by an interarrival time generator. Interarrival times for subsequent queue elements are the outputs from the previous stages in the simulation. Service times for each queue element are generated by a service time generator. Each box in the figure represents a task in the simulation that can execute in parallel.

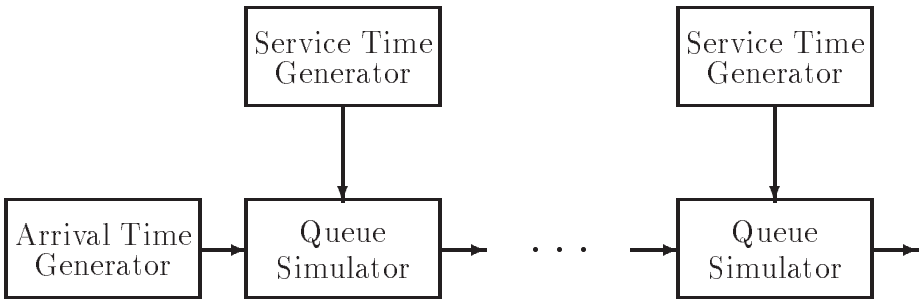

Figure 6: Structure of the queueing simulation

The arrows in Figure 6 represent a sequence of floating point values. We will represent a sequence by a list of `cells` whose type declaration is in Figure 7. Notice that the fields for `cell` are declared `sync`.

The code for the arrival generator and service time generator is shown in Figures 8 and 9 respectively. These routines simply generate a list of cells. The class `randm` is a random number generator. With the exception of the `sync` type modifiers, these routines are no different from their C++ counterparts.

The implementation of the queue element is shown in Figure 10. Note what happens in the `while` loop if the simulator gets ahead of either the service time generator or the interarrival time generator. The pointers to these lists are updated by assigning them values from the `next` component of a `cell`. Since the `next` field is a `sync` pointer, this assignment will block

```
inline float max(float a, float b) { return ((a>b) ?  a :  b);};

struct cell {
  sync float value;
  sync cell * next;
  cell(float t) {value = t;}    // initialize new cell
};
```

Figure 7: Data type declarations for queueing simulation

```
void generate_arrivals(int n, float mean, randm rand, cell * sync p)
{ // n > 0
  // p is set to be a list of n arrival times with mean value "mean."

  cell * q = p;
  float t = 0.0;
  int i;

  for(i=0; i<n; i++) {
    t = t+rand.generate(mean);
    q->next = new cell(t);
    q = q->next;
  }
  q->next = (cell*) NULL;
};
```

Figure 8: Routines to generate interarrival times in queueing simulation

```
void generate_service(int n, float mean, randm rand, cell * sync p)
{
  // n > 0
  // p is set to be a list of n service times with mean value "mean."

  cell * q = p;
  int i;

  for(i=0; i<n; i++){
    float t = rand.generate(mean);
    q->next = new cell(t);
    q = q->next;
  }

  q->next = (cell* sync)NULL;
};
```

Figure 9: Routines to generate service requirements in queueing simulation

```
void simulate_queue(cell* sync a, cell* sync s, cell* sync d)
{
  // a and s are nonempty input lists of arrival times
  // and service times respectively, and d is set to
  // be the nonempty list of departure times.
  // The length of d is the max of the lengths of a and s.

  float t = 0.0;  // t is the time of the first departure
  cell * pd = d;
  cell * pa = a->next;
  cell * ps = s->next;

  while((pa != (cell * ) NULL) && (ps != (cell * ) NULL)){
    t = (ps->value) + max(t,pa->value);  // Next departure time
    pd->next = new cell(t);
    pa = pa->next;
    ps = ps->next;
    pd = pd->next;
  };
  pd->next = (cell * sync) NULL;
};
```

Figure 10: Routine to simulate a queue

```
int main() {
  const int num_servers = 37;
  const float mean_interarrival = 1.0;
  randm rand[num_servers+1];
  float mean_services[num_servers];
  cell * sync a = new sync cell(0.0);
  cell * sync s[num_servers] , * sync d[num_servers];

  // Initialize mean_service time distributions and random number
  // generators here.

  parfor(int i = 0 ; i < num_servers; i++) {
    s[i] = new cell(0.0); d[i] = new cell(0.0);
  }

  par {
    generate_arrivals(n, mean_interarrival, rand[0], a);
    simulate_queue(a,s[0],d[0]);
    generate_service(n, mean_services[0], rand[1], s[0]);

    parfor(int j = 1; j <  num_servers ; j++)
      par {
        simulate_queue(d[j-1],s[j],d[j]);
        generate_service(n, mean_services[j], rand[j+1], s[j]);
      }
  }
}
```

Figure 11: Main program for queueing simulation

until the next `cell` structure in the list has been created. The queue elements self-synchronize themselves. Again note, that with the exception of the `sync` type modifiers, this is plain C++ code.

Figure 11 is the main driver for the queueing simulation. The first task is to create a set of cells that are to be used to connect the simulators, service time generators and arrival time generators together. The main part of this routine is responsible for creating the queue elements and service time generators. This is done by a `par` block. Within the `par`, the initial arrival time generator, service time generator and first queue element are created and connected together via shared cells. The rest of the queue is constructed in parallel by a nested `parfor` statement. Each loop body is itself a `par` block that creates a queue element and a service time generator and connects them together using shared references to a `cell`.

4 Conclusions

In this paper, we introduced Compositional C++ as a demonstration of how compositional programming can be supported in an imperative programming language. The advantages of CC++ include:

- CC++ is based on a popular programming language. The advantages of C++ are advantages of CC++ as well. These include strong typing, data abstraction and object-oriented programming. Object-oriented design methodologies [7] that facilitate the construction of large scale object-oriented programming systems can be applied to software written in CC++ as well.

- CC++ is C++ with a *small* number of extensions. C++ programmers can learn CC++ within an hour.

- CC++ provides a mechanism for parallel programming, not a policy. Programmers can develop different types of parallel programs in CC++: deterministic or nondeterministic, single program multiple data (SPMD) or multiple instruction multiple data (MIMD). Programmers can use CC++ libraries that implement virtual channels, semaphores and monitors. Because of this flexibility, CC++ is suited for both scientific and reactive applications.

- CC++ was designed to support formal methods. This focus helps not only in the design and implementation of correct programs, but can be used to aid in testing and debugging as well.

The principles used in the design of CC++ can be applied to other programming languages as well. Other compositional languages being investigated include Fortran and Ada.

CC++ is currently implemented on shared-memory parallel computers and on uniprocessor workstations. We anticipate having an implementation available on distributed-memory parallel supercomputers and networks of workstations shortly.

5 Acknowledgment

The research on CC++ object libraries for concurrent computation is funded by DARPA under grant N00014-91-J-4014. The research on compositional concurrent notations is funded by the NSF Center for Research on Parallel Computing under grant CCR-8809615. Support of CC++ application development is funded by the Office of Naval Research under grant N00014-89J-3201.

References

[1] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[2] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Arvind and R.E. Thomas. I-Structures: An efficient data structure for functional languages. Technical Report TM-178, MIT, 1980.

[4] William C. Athas and C.L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, August 1988.

[5] Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A system of object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[6] Ulla Binau. Real good stuff in CC++. Technical Report Caltech-CS-TR-92-11, California Institute of Technology, 1992.

[7] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[8] K. Mani Chandy and Carl Kesselman. The CC++ language definition. Technical Report Caltech-CS-TR-92-02, California Institute of Technology, 1992.

[9] K. Mani Chandy and Carl Kesselman. The derivation of compositional programs. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.

[10] K. Mani Chandy and Carl Kesselman. Mutual exclusion in a token ring in CC++: Program and proof. Technical Report Caltech-CS-TR-92-01, California Institute of Technology, 1992.

[11] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[12] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Bartlett and Jones, 1991.

[13] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[14] Ian Foster. Information hiding in parallel programs. Technical Report MCS-P290-0292, Argonne National Laboratory, 1992.

[15] Ian Foster and Stephen Taylor. *STRAND: New Concepts in Parallel Programming*. Prentice Hall, 1989.

[16] Andrew S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Computer Science Report TR-91-07, University of Virginia, 1991.

[17] Carl E. Hewitt and Henry Baker. Actors and continuous functionals. In *Proceedings IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–387, August 1977.

[18] David Kuck, Daniel Gajski, et al. A second opinion on data flow machines and languages. *Computer*, 15(2):58–69, February 1982.

[19] Max Lemke and Daniel Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. Arbeitspapiere der GMD 611, Gesellschaft Für Mathematik und Datenverarbeitung MBH, February 1992.

[20] J. McGraw et al. SISAL: Streams and iteration in a single assignment language, language reference manual, version 1.2. Technical Report M-146, LLNL, March 1985.

[21] Jayadev Misra. Personal Communication, 1992.

[22] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(1):319–340, 1976.

[23] Amir Pnueli. Personal Communication, 1992.

[24] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition edition, 1991.

[25] Bjarne Stroustrup and Jonathan Shopiro. A set of C++ classes for co-routine style programming. In *Proceedings of the USENIX C++ Workshop*, November 1987.

[26] Andrew A. Chien Waldemar Horwat and William J. Dally. Experience with CST: Programming and implementation. In *SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.