# High Performance Parallel and Distributed Computation in Compositional CC++

Carl Kesselman
The Beckman Institute
California Institute of Technology
carl@compbio.caltech.edu

**Abstract**

C++ was designed to support a wide range of programming styles. A parallel extension to C++ should support parallelism in all the different programming styles that a C++ program might exploit as well as support a broad range of parallel programming paradigms, such as message passing, active objects and shared memory concurrency. Compositional C++ (CC++) is a parallel C++ that was designed with these goals in mind. In this article, we motivate the design of CC++ and provide a brief overview of the language. We will illustrate how CC++ can be used to define a reusable parallel paradigm libraries and describe how one parallel application used CC++ language features to combine different types of parallelism in a single application.

## 1 Overview

C++ is a widely used programming language that is finding increased use in complex, irregular, high-performance applications. For this reason, a parallel version of C++ is desirable. While C++ *supports* object-oriented programming, the language was designed to enable a range of different, non object-oriented programming styles. A good case in point is the generic algorithm programming style used extensively in the definition of the Standard Template Library [1]. In designing a parallel version of C++, it is important to enable parallel execution in all types of C++ programs, not just the ones that use the object-oriented programming style.

Additionally, there are many different parallel programming styles including message-passing or active objects cincurrency and shared memory multi-threading. Complex, high-performance applications may require the use of one, or more, different parallel programming paradigms. For this reason, a parallel C++ should support a rich assortment of parallel programming styles.

Compositional C++ (CC++) [2] was designed to address these issues. CC++ consists of six language extensions that augment C++ with a well defined parallel semantics. The constructs were not designed to provide any particular parallel programming style. Rather, they were designed so that they can be combined with each other and the rest of the C++ language to create reusable parallel paradigm libraries which implement a specific parallel programming paradigm.

The work described in this paper is supported by DARPA under contract DABT63-95-C-0108.

## 2 CC++ Language constructs

CC++ defines an explicitly concurrent language model with a global namespace. The language constructs enable explicit control over concurrency, locality and synchronization. In designing a parallel language, one often must trade off issues of safety expressibility and efficiency. CC++ was designed from the perspective that C++ is an inherently unsafe language, in which some operations, such as pointer casting can be used erroneously causing a program to fail. While the existence of unsafe operations can complicate the process of program development, programmers often turn to C or C++ because they require a high degree of control over the execution of their program. For these reasons, the CC++ language design takes the position that we cannot in general sacrifice performance or control for safety, that is, it is possible to use some CC++ constructs incorrectly, just as in C++.

### 2.1 Concurrency in CC++

Three control structures are used to introduce concurrency into an application: the parallel block, parallel loop and function spawn. The form of a parallel block is:

$$\texttt{par \{ statement}_1\texttt{; ...; statement}_n\texttt{; \}}$$

This parallel block causes new threads of control to be created to execute each statement, and that the threads execute in a fair, interleaved manner [3]. The `par` block statement terminates when all of the enclosed statements terminate. The parallel for loop and function spawn work in a similar manner. Within a parallel statement, all memory is shared. It is the responsibility of the programmer to prevent data races and provide required synchronization between concurrently executing statements. Note that unlike the `conc` block in IC++ [4], CC++ makes no attempt to preserve sequential semantics in parallel blocks. While the `conc` semantics can reduce the possibility of introduction of errors caused by race conditions, it limits the range of parallel programs that can be expressed.

The parallel block and parallel for loop provide a structured form of concurrency in that these statements do not terminate until the enclosed statements terminate. Alternatively, the `spawn` statement creates unstructured concurrency. This statement creates a new thread of control to execute the specified function and continues execution without regard to the state of the spawned function.

### 2.2 Synchronization in CC++

CC++ provides two different synchronization mechanisms: `atomic` functions and `sync` variables. A function can be made atomic by adding the keyword `atomic`:

```
atomic test_and_set(x);}
class int_queue {
public:
    atomic int dequeue();
    atomic enqueue(int);
```

within an instance of a class, only one `atomic` member function may execute at a time. If the atomic function is a regular `C` function, then it may not execute concurrently with any other `C` type atomic function.

Note that there is no prohibition on atomic and non-atomic functions executing concurrently. This is quite different from concurrent object-oriented programming languages such as ABCL [5] in which an object has a single thread of control. Requiring a single thread within an object would restrict the types of parallel programs that could be expressed. If single-threaded semantics are desired, this can easily be achieved by making every member function of an object `atomic`.

More detailed control of synchronization can be obtained by the use of `sync` variables. Any C++ data type can be declared to be a synchronization or `sync` type:

```
sync int x;  // sync integer
sync float * sync y; // sync pointer to sync float
```

Any attempt to read a `sync` variable before it has been initialized will cause the reader to block.

Standard synchronization constructs such as locks and condition variables are easily implemented using `sync` variables in combination with `atomic` functions. For example, a mutual exclusion variable (mutex) can be implemented by using a list of `sync` variables to cause a lock request to block and atomic functions to sequence requests to lock and unlock the mutex. In addition, `sync` variables enable the design of fine grain, customized synchronization operations. It has been demonstrated that use of synchronization variables can have a significant performance advantage over other synchronization mechanisms [6].

### 2.3 Locality in CC++

CC++ defines an explicit, two level model in which references are distinguished by being local or remote. A reference to a remote object is designated by the type modifier `global`. For example:

```
int * global gPtr;
```

declares a pointer that can refer to a local or non-local integer. The keyword `global` is used to indicate to the programmer that operations on a global pointer may be more expensive then the same operation on a local pointer. Global pointers make the cost of data-access explicit, providing a means by which a programmer can manage application locality. Global pointers may also be found in the Split C [7] parallel programming language.

Any operation that may be performed on a pointer can also be performed on a `global` pointer, including calling a member function of an class, resulting in a remote procedure call. Operations on global pointers can cause communication to take place. However, the CC++ compiler is able to manage the details of the communication:

```
int * global x;
int y;
class C {
int foo(int);
} * global cPtr;

*x=y; // put the value of  y in a remote x.
y=*x; // store the value of a remote x in y.

// Call foo remotely. cPtr and x do not
//  have to be in the same place.
y = cPtr->foo(*x);
```

Non-local references are initially created by the introduction of *processor objects* into a CC++ program. A processor object differs from other C++ objects in that it defines a new, unshared address space; global and static variables are local to an instance of a processor object. Because a processor object defines a new address space, it provides a mechanism by which existing C++ code can be migrated into a parallel application.

A processor object can be created via the `new` operator, just like any other C++ object. While the placement argument to the `new` operator can be used to specify where in an address space an object is to be located, the placement argument for a processor object `new` specifies where in "process space" the processor object is located. A processor object enables an application to separate the logical structure of an application from the physical environment on which the application runs. For example, since many different processor objects can be mapped to the same physical node, we can structure a computation as a square grid of processor objects, regardless of the number of actual nodes available.

## 3 Programming Paradigms in CC++

A central goal in the design of CC++ was to support the efficient implementation of parallel programming styles via reusable paradigm libraries. To demonstrate this ability, we have designed and implemented a standard CC++ library that contains many common parallel programming paradigms. Programming styles supported by this library include:

- POSIX style thread operations such as thread creation, mutual exclusion variables, and condition variable.
- a parameterized channel class abstraction that provides typed point-to-point communication. This library does not introduce any additional copy operations over those required by the underlying communication layer.
- A global operations library that supports reductions and barriers over arbitrary collections of processor objects.
- An SPMD programming environment which enables easy porting of existing message passing codes.

The SPMD library provides an interesting demonstration of the utility of code reuse. This library defines a traditional message passing environment in CC++. In its implementation, it combines the channel class library, the global operations library and several other components of the standard CC++ library.

## 4 Building Large Scale Parallel Systems

To illustrate how the features of CC++ enable the development of large parallel applications, we will present an overview of an application that was developed using CC++. This application, called CNEPH, processes inferred and visual satellite data to locate the position and elevation of clouds. This information is then geolocated onto a terrain map and the results rendered in 3D on an Immersdesk, a immersive virtual reality device. A detailed description of this application, which is completely written in CC++, can be found in [8].

There are three types of processing that go on within CNEPH:

- input processing,
- cloud detection, and
- display processing.

Input processing has access to the image data from the satellite downlink. Cloud detection takes the raw image data produced by input processing and determines cloud location and elevation. Cloud detection can take place in parallel. Finally, the results of the detector are converted into polygons and rendered in 3D by the display processor.

Each type of activity in CNEPH is performed by a different CC++ processor object type. By using processor objects, we are able to logically structure the application and we have the ability to cleanly map specialized types of processing, such as input processing or display processing onto specialized computing resources.

When CNEPH starts it creates an input processor object, a collection of cloud detector processor objects and a display processor object. The input and display processor object are typically located on specialized machines as they require either the satellite downlink or a graphics engine. The cloud detector processor objects are usually created on nodes of a parallel computer. We then make a member function call into the input processor object passing in a global pointer to each detector object. Likewise, we make a member function call in each detector object, passing in a global pointer to the display object. Since global pointers can be used as arguments to routines, just like regular **C** pointers, the process of setting up the computation structure is straight forward.

CNEPH exploits different types of parallelism. The image data can be processed in blocks. To mask the I/O latency of reading the image data, CNEPH pipelines the communication of image data into the detection nodes. A double buffer object is build to ensure that an unprocessed image block is waiting on the detector node. A `sync` variable is used to enure that the detector does not run ahead of the buffer object.

The detector nodes act as a task farm, processing the input data blocks as it comes from the input object. When a detector is done with an image block, it must pass the result on to the display object. This is done by invoking a member function in the display object. This member function integrates the newly processed block into a polygon database. The display object exploits shared memory parallelism. There are two threads of control in the display object. One thread extracts polygons from the image data while the second thread runs the graphics libraries that render and display the polygons.

As can be seen from this description, CNEPH is a complex application that combines different types of parallelism, different types of computer architectures and diverse component functionality. In developing CNEPH, CC++ played a significant role simplifying the design and implementation process.

## 5 Summary

The CC++ language consists of a small number of extensions. These extensions have proven to be rich enough to address the requirements of a wide assortment of parallel applications. The current CC++ compiler is available and is currently being used by a number of different organizations and projects. We are currently developing a second generation compiler that is based on the ANSI C++ language specification. In the future, we will be developing optimization techniques for CC++ and are studying the performance of CC++ paradigm libraries. Additional information and the CC++ compiler can be found at:

`http://www.compbio.caltech.edu/ccpp`.

## References


[1] David R. Musser and Atul Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison-Wesley, 1996.

[2] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[3] K. Mani Chandy and Carl Kesselman, The Derivation of Compositional Programs, Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming, 1992, MIT Press

[4] Concurrent Systems Architecture Group. The ICC++ reference manual. Concurrent Systems Architecture Group Memo, May 1996.

[5] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[6] J. Thornley, A Parallel Programming Model with Sequential Semantics, Ph.D. Thesis, Caltech, May 1996.

[7] D. Culler, A Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta and T. von Eicken, K. Yelick. "Parallel Programming in Split-C", "Proc. Supercomputing '93", 1993

[8] C. F. Kesselman, C. Lee, and S. Schwab Near-real-time Satellite Image Processing:Metacomputing in CC++, Computer Graphics and Applications, ACM, 1996