C= Parallel C/C++ Language Extension

C= Specification and Reference Manual

Revision 0.7

2013

Contents

Overview

C= (pronounced 'See Stripes') is a Parallel C/C++ Programming Language Extension. It is designed to cover all typical parallel programming paradigms and complement or even substitute existing parallel language extensions and libraries by providing a single parallel language most naturally connected with the C/C++ syntax and semantics.

The principles and constructs of C= including flexible task-thread semantics, parallel visibility scopes, object-free synchronization, and wake-wait task interaction paradigm are conceived to be easily adopted by other procedural and object-oriented languages, compiled or interpreted.

In brief, the main idea of C= is to motivate parallel thinking, let programmers focus on writing parallel programs in the first place, rather than do the job twice by implementing a sequential program and then trying to multithread it (though the latter approach, inevitable in many cases, is also elegantly supported).

Currently, the C= language extension is implemented as a C library and does not imply specific compiler support, nor radical changes to existing software building processes.

In this document we're going to share all necessary information to assist a C/C++ programmer in smooth transitioning to parallel programming in C= by providing a high-level language specification, explaining underlying parallel processing algorithms, furnishing various code samples, and commenting on best programming practices.

Getting Started

Let's begin with the traditional 'Hello, world' example re-written in C= to print the greeting in parallel, and see how naturally understandable it is to a native C/C++ programmer. This example will also help us introduce and illustrate the basic (and most important) concepts of task creation, execution, identification, synchronization and data sharing.

```
void salute()
{
    parallel()
    {
        int idx = pix();

        serial()
        {
            parallel(3)
            {
                printf("Hello, world, from task %d-%d\n", idx, pix());
            }
        }
    }
}
```

As you can see, the *salute* function contains two nested regions of parallel execution. Each region of parallel execution is formed by the *parallel* operator, whose body, that is, every statement within the *{}* scope, is (not surprisingly) executed concurrently by multiple tasks or threads (please read further to learn how C= differentiates between those two terms). It should be noted here that only the thread that entered the *parallel* operator (initial thread) can continue execution beyond the parallel scope, any other task/thread created/engaged by the *parallel* operator cannot go over the closing brace.

The first *parallel* operator (without operands) executes its body in the number of tasks that is equal to the number of available processors.

The *pix* built-in function (an acronym of 'Parallel IndeX') returns the integer index of the current task. That index is stored in the *idx* variable, which is private to the tasks of the current parallel region, that is, each parallel task possesses its own copy of *idx* initialized with each task's own parallel index. Note that the initial task/thread is always assigned index 0.

The *serial* operator (not surprisingly, again) serializes execution of the statements within its scope, in other words, it ensures that its body is executed by only one task of the outer parallel region at a time.

Inside the serial region, there's the second parallel region, though this time the number of concurrent tasks is explicitly specified as an operand of the *parallel* operator.

Naturally, the *pix* function, when used in the second parallel region, returns indices of the second region's tasks within [0..2] range. Another natural assumption is that the value of the *idx* variable set up by a task from the first parallel region is visible to all tasks of the second, nested region.

Now we're equipped to understand the entire operation of the above code sample: it creates a parallel task per processor, saves each task index, and then for each task, one by one, creates three more tasks, and prints the following (on a dual-processor system):

```
Hello, world, from task 1-1
Hello, world, from task 1-3
Hello, world, from task 1-2
Hello, world, from task 0-1
Hello, world, from task 0-2
Hello, world, from task 0-3
```

The actual sequence of prints may be arbitrary, but the output from the same processor is guaranteed to be grouped together (by the *serial* operator).

Summing up, the primary concept of C= is to introduce parallel control flow operators, rather than just means of data exchange, to cover the widest possible range of parallel programming paradigms, and inherit the concept of data visibility scopes from sequential languages to define standard rules of data sharing and parallel access.

See the next section to learn how the above ideas are supported by internal algorithms of parallel execution.

Parallel Execution Management Rules and Semantics

First of all, let's define the following terms: *task* and *thread*.

A *thread* is a system-level execution entity. A program, even a sequential one, is always executed within a context of at least one thread.

A *task* for the purposes of C= language is a portion of work that can be executed by a thread. Each task is represented by an execution state, which is hardware and system-dependent and is opaque to a C= programmer.

Let's track the execution of the parallel operator step-by-step:

```
parallel()
{
    ...
}
```

1. The thread executing a sequential program encounters the *parallel* operator.
2. It saves the current execution state, plus computes and saves the total number of tasks that need to be formed out of the saved state and executed.
3. The thread fetches the first task (index 0, so-called *initial task* or *thread*) and signals to a pool of threads from C= runtime library to start fetching and executing tasks.
4. Each thread of the C= runtime thread pool fetches a task and executes it.
5. The procedure repeats until there's no task left.
6. The task execution state ends at the closing brace of the *parallel* operator, so no task/thread can continue execution beyond the parallel scope, except for the *initial thread* that entered the *parallel* operator.
7. The initial thread continues executing the sequential program, while pooled threads wait on a semaphore.

As becomes evident from the above explanation, such a scheme of parallel execution management automatically ensures efficient processor utilization and dynamic work balancing and eliminates thread conflicts due to work stealing: each thread *forms* a new task from the saved execution context independently, rather than stealing the work from other threads.

But in many cases there is a need for a dedicated thread to process asynchronous events, pre-load data, etc. C= provides all necessary means for that inside the same *parallel* operator: whenever a task tries to obtain its system-unique identifier (to be shared with other threads and to be used for thread synchronization), the C= runtime 'promotes' the task to an independent thread, and adds a new thread to the thread pool to continue fetching and executing other parallel tasks.

C= provides special functions to asynchronously wake such an independent thread and to wait for a wakeup signal, thus defining an easy-to-follow parallel thread interaction paradigm (please see the accompanying code samples).

The code sample from the previous section makes use of the *serial* operator. The operator, in addition to its reduced form (as shown in the above sample), allows for specifying a pointer to the actual memory object which has to be protected from being accessed in parallel. In effect, data structures of sequential programs do not need to be redesigned in order to accommodate for various locks and other synchronization objects. On the contrary, the *serial* operator of C= enables object-free synchronization and saves both memory and programmers' efforts.

Another important feature illustrated in the previous section's example is the use of variable visibility scopes to differentiate between shared and private variables. C= declares the following sharing rules:

(1) All variables within a parallel scope are private.
(2) All global variables (beyond a function scope) are shared by reference; that is, each parallel task accesses the same physical object while operating on such a variable.
(3) All local (automatic) variables located within the function scope but beyond (upper than) the parallel scope are shared by value; that is, each parallel task can read the same value when accessing such variables. No assumption can be made as to whether the values are duplicated or parallel tasks access the same data location. Hence it is recommended to update shared variables through shared pointers and initialize those pointers prior to entering a parallel region – it is a safe, least-common-denominator practice to meet possible limitations of different implementations of C=.

See the next section on syntactic and semantic details.

Language Specification

This section recites C= language specification, though not in the formal way of ISO standards but rather in a more relaxed form, which nevertheless provides enough information to a C/C++ programmer on C= syntax.

Hereinafter C= normative syntax elements are printed in black, whereas syntax elements specific to the current implementation are shown in gray. Those gray elements will become implicit in complier- or preprocessor-based implementations.

An operator to execute enclosed statements *{}* in parallel by the number of tasks specified by *expression*; *expression* comprises a valid C/C++ expression that evaluates to an integer value, or an empty string; in case *expression* is empty, the number of tasks in the parallel region is automatically set equal to the number of available processors:

```
parallel(expression)
{
}
```

An operator to execute enclosed statements *{}* atomically, that is, sequentially with respect to other tasks simultaneously entering this operator; *expression* comprises a valid C/C++ expression that evaluates to a void pointer, or an empty string; in case *expression* is empty, the operator serializes execution of all tasks of the current parallel region; otherwise, the operator serializes execution of every task that executes the operator with the same expression (that is, is going to reference the same memory object):

```
serial(expression)
{
}
```

Note that any combination of nested *serial* operators is allowed. As a good practice, use the same nesting order of serial operators when synchronizing memory accesses from different functions.

A non-blocking *trial serialization* operator to execute statements in *try{}* atomically in case no other task is executing the *serial* operator with the same *expression*; otherwise, execute statements in *else{}* block:

```
try(serial(expression))
{
}
else
{
}
```

Please note that no control transfer to or from the middle of the *parallel* or *serial* operator scope is allowed. That effectively disallows the use of `return`, `goto`, `setjmp`, and `longjmp` statements inside the operator scope and forces exception handling not to cross the operator scope as well.

A built-in function to obtain the index of the current parallel task within the current parallel region; index 0 is always assigned to the initial task/thread in whose context the parallel operator was invoked:

```
int pix();
```

A built-in function to obtain the number of parallel tasks in the current parallel region:

```
int pno();
```

A built-in function to obtain the parallel ID of the current thread; in the current C= implementation causes the C= runtime to promote the current task to an independent thread; returns 0 in case of error, for instance, insufficient system resources to convert the current task to an independent thread; it is a good practice to call this function in all tasks of a parallel region unconditionally:

```
void* pid();
```

A built-in function to wake a thread of the specified parallel ID:

```
void wake(void* pid);
```

A built-in function to wait for a wakeup signal infinitely:

```
void wait();
```

An overloaded built-in *wait* function to wait for a wakeup during the specified number of milliseconds; returns 0 in case the thread is woken up by the *wake* function; returns a non-zero value if timed out:

```
int wait(int milliseconds);
```

An explicit barrier-function to be used at the end of a parallel region; mutually exclusive with the *break* operator; will become implicit in compiler-based C= implementations:

```
barrier();
```

An operator to skip the implicit barrier of a parallel region; mutually exclusive with the *barrier* function; will merge with the standard C/C++ keyword in compiler-based C= implementations; currently, in the library-based implementation, is also used to exit serial regions:

```
break();
```

A keyword to ensure a shared variable is not stored in a register; will become implicit in compiler-based C= implementations (and can be currently replaced with the `volatile` keyword):

```
shared
```

Utility macros to distribute an integer range *N* among all tasks of the current parallel region; the functionality of these macros can be replaced by a regular C= code (see programming examples), they are not an inherent part of the language and are provided for convenience:

```
CPX_RANGE_START(N);
CPX_RANGE_END(N);
```

An operator to execute enclosed statements *{}* concurrently by processors and co-processors or remote systems (this operator must be used inside a *parallel* operator; and the current C= implementation allows only one *coload* operator per *parallel* operator):

```
coload(comma-separated-descriptor-list)
{
}
```

wherein *comma-separated-descriptor-list* comprises the following descriptors:

`cpx_in(type, pointer, optional size),`

> which specifies a *pointer* to an array of certain *type* and *size* that can be read from inside the *coload* operator;

`cpx_out(type, pointer, optional size),`

> which specifies a *pointer* to an array of certain *type* and *size* that can be written from inside the *coload* operator;

`cpx_inout(type, pointer, optional size),`

which specifies a *pointer* to an array of certain *type* and *size* that can be read and written from inside the *coload* operator;

`cpx_const(value),`

which specifies a single input *value*;

`cpx_warp(number),`

which specifies the *number* of co-processor work items that can share the same memory;

`cpx_cache(size),`

which specifies the *size* of memory shared by the co-processor work items. Note that all data declared inside a parallel scope above the *coload* operator are mapped to the co-processor's local memory shared by work items;

`cpx_range(range),`

which specifies the integer *range* to be distributed among processors and co-processors;

`cpx_error(pointer),`

which specifies a *pointer* to an integer variable to receive a non-zero error code in case of a co-processor's error, from which the C= runtime failed to recover. The C= runtime tries to roll back to processor-based computations whenever a co-processor fails to execute the *coload* operator, but there may be situations requiring assistance from a programmer (e.g., when the data or execution state had already been modified by the time of a failure).

An explicit co-processor barrier-function to ensure multiple co-processor work items that share the same memory have the same memory state; will become implicit in compiler-based C= implementations:

```
cpx_sync();
```

See the next section for programming examples and implementation-specific information.

Programming Examples

This section illustrates C= programming in most typical examples and also describes extra rules and limitations imposed on C= programmers by the current non-compiler-based implementation, as well as best practices of C= programming.

For an extensive set of C= code samples and detailed comments, please refer to the accompanying readme.cpp file.

Strictly speaking, the example furnished in the 'Getting Started' section is written in pure C= and needs to be slightly modified as shown in readme.cpp to meet the limitations of the current library-based implementation.

Let us study the most typical case of data decomposition in the following example that multiplies two arrays, element-by-element, and prints the accrued sum of the products:

```
const int N = 0x1000;
float a[N], b[N];

void decompose()
{
    /// this variable is shared by value
    float sum = 0;
    /// shared by value and since it is a pointer
    /// all its copies point to the same location
    /// shared keyword inserted before psum ensures that both sum and
    /// psum aren't register variables, and all
    /// *psum operations will not be removed by the compiler
    float* shared psum = &sum;

    parallel()  /// execute the code below by
    {           /// the number of tasks = the number of processors
        /// private variables
        /// some compilers pre-compute constants,
        /// keep them in registers,
        /// and do not do the actual initialization;
        /// so there may be a need for explicit initialization
        /// unless C= is supported natively
        volatile float zero = 0;
        /// this should not be volatile to allow for optimizations
        float s = zero;
        /// use pre-defined macros to divide the data
        /// into non-overlapping regions (static decomposition of work)
        /// each task automatically receives its portion of work,
        /// not necessarily equal
        for(int i = CPX_RANGE_START(N),
                end = CPX_RANGE_END(N); i < end; i++)
        {
            /// do multiplication normally
            s += a[i] * b[i];
        }
        /// serialize all tasks of this parallel region
        serial()
        {
            /// reduce the local sums into one shared sum
            *psum += s;
            /// exit the serial region (will become implicit)
            break();
        }
        /// ensure the initial task/thread that entered the parallel
        /// region does not exit before all other tasks complete
        ///(will become implicit)
        barrier();
    }
    printf("Sum = %f\n", sum);
}
```

Now let's illustrate dynamic work balancing in the example of Fibonacci numbers computation:

```
/// compute the Fibonacci number of order n
int fibonacci(int n)
{
   if(n < 2)
   {
      return n;
   }
   return fibonacci(n - 1) + fibonacci(n - 2);
}

void computeFibNumbers()
{
     /// allocate a vector of Fibonacci numbers to compute
     std::vector<int> a;
     /// set up a shared pointer to access the vector
     std::vector<int>* shared ap = &a;

     /// initialize the vector
     a.push_back(41);
     a.push_back(24);
     a.push_back(26);
     a.push_back(42);

     /// set up the output vector and the shared pointer to access it
     std::vector<int> v;
     std::vector<int>* shared vp = &v;

     /// pre-allocate the output space
     v.insert(v.begin(), (const int)a.size(), 0);

     /// create as many tasks as there're elements in the input vector
     /// C= runtime ensures to run one task per processor at a time
     /// and fetches new tasks for execution automatically,
     /// so that the work is evenly balanced between processors
     parallel(a.size())
     {
         /// get the current task's index (just an optimization)
         int i = pix();

         /// compute the task's Fibonacci number and save it
         (*vp)[i] = fibonacci((*ap)[i]);

         barrier();
     }
     /// print the results
     std::cout << "Fibonacci: ";

     for(std::vector<int>::iterator i = v.begin(); i < v.end(); i++)
     {
         std::cout << *i << " ";
     }
     std::cout << std::endl;
}
```

To avoid possible limitations of different C= implementations it is recommended to adhere to the following programming practices:

- Access a shared variable via a shared pointer and explicitly specify the pointer as such by inserting the *shared* keyword immediately before the pointer's name.
- Use the *volatile* modifier to ensure a variable outside of the parallel scope is not kept in a register (in case of reading the variable from the parallel region).
- Do not obtain the address of an outside variable (using & operator) from a parallel region.
- Do not jump into or out of a parallel or serial region, do not return from the function when executing code inside a parallel or serial region, handle all exceptions within the parallel/serial operator scope.
- Please keep in mind that not all existing runtime functions are thread-safe, exercise care when invoking them inside a parallel region.
- Avoid declaring large automatic (local) arrays in functions containing parallel regions as that may have a negative performance impact in case of non-compiler-based C= implementations.

To switch between parallel and sequential versions of your program quickly, you may want to design your code (when and where applicable, of course) for an arbitrary number of tasks and use an explicit argument in the *parallel* operator: `parallel(x){}`, so that the argument can be easily redefined: `#define x 1`.

And finally, an example illustrating the ease of heterogeneous programming in C=:

```
const int N = 0x100011; float a[N], b[N], c[N];

void sumVectors()
{
    /// run concurrently on all CPUs
    parallel()
    {
        /// run concurrently on all CPUs and GPUs
        coload(
               /// use arrays a and b as input
               cpx_in   (float*, a),
               cpx_in   (float*, b),
               /// use array c as both input and output
               cpx_inout(float*, c),
               /// the total amount of work is N elements
               cpx_range(N))
        {
            /// use standard macros for data decomposition
            /// GPUs execute loop iterations in parallel,
            /// both CPUs and GPUs compete for portions of data
            for(int i = CPX_RANGE_START(N); i < CPX_RANGE_END(N); i++)
            {
                /// the heavier computations are used in the loop body
                /// the greater performnace can be gained on GPUs
                c[i] += a[i] + b[i];
            }
        }
        barrier();
    }
}
```

Note that the same program can operate concurrently on any reasonable combination of processors and co-processors (even when no co-processor is available) without the need for redesign or modification.

See the next section on how to start using C= in software projects.

Usage Model

The current implementation of C= runtime library is compatible (subject to the accompanying disclaimer – see legal notice above) with Microsoft[1] Windows and Linux operating systems (please consult the accompanying readme.txt file for the list of OS versions) and supports IA32 and Intel 64/AMD64 processor architectures (including Intel Many Integrated Core architecture).

The current implementation of C= language extension implies the use of a C/C++ compiler that supports variadic (variable-argument) macros.

C= package does not require installation and is always ready to use, however, for the sake of convenience, the package can be registered with Microsoft Windows operating system and Microsoft VisualStudio C/C++ integrated development environment as explained in the accompanying readme.txt file.

C= package can be used as an ordinary C library by including *C=.h* header file and linking your project with *C=.lib* file as appropriate for target hardware architecture. To run your resulting executable, please make sure the appropriate *C=.dll* can be found on the system DLL loading path.

To enable the use of co-processors, e.g., General-Purpose Graphics Processing Units (GP-GPUs) with OpenCL or CUDA support, it is required to generate execution kernels from your C= code by applying *c2cl.bat* and/or *c2cu.bat* commands to all source files containing *coload* operators, and then copying the resulting files (*.ocl* and *.ptx*) next to your executable binary (refer to the accompanying readme.txt file for instructions).

C= runtime will then be able to pick up appropriate kernels automatically. Note that generating a *.ptx* file requires NVIDIA CUDA SDK to be pre-installed.

To make C= programs run on distributed systems, generate kernels for remote execution of coload operators by applying *c2net.bat* commands to all source files containing *coload* operators, and then copying the resulting file (*.dll*) next to your executable binary.

In order to set up a remote execution environment, run *coagent.bat* on a remote system and specify a path to a shared folder to be used for data exchange. To clean up the shared folder and force remote agents to re-arbitrate for the shared folder resource, run *coclean.bat* specifying the shared folder path as a parameter (refer to the accompanying readme.txt file and for further instructions).

Setting up *CPX_NETLOAD_SERVER* environment variable on your local system to the shared folder path enables all locally executed C= programs distribute their workloads to remote systems. Note that the same environment variable can be set up on remote

[1] All names and brands referenced in this document may be claimed as the property of their respective owners

systems, as well, pointing to different folders and thus establishing a hierarchy of remote agents to better adapt to your computation tasks' needs and system capabilities.

In case the C= package is registered with Microsoft VisualStudio, vesrions 8 or 9, it is enough to add "`#include <C=.h>`" directive to your source files, and the linking will be performed automatically, even for pre-existing projects.

C= package has no specific dependencies on language runtime or other libraries, which implies its compatibility (subject to the above disclaimer) with other parallel programming solutions (for instance, Intel Threading Building Blocks, OpenMP, and Microsoft Concurrency Runtime). However, there may be occasional conflicts as keywords defined by different parallel programming extensions may coincide: thus OpenMP standard makes use of the *parallel* keyword: `#pragma omp parallel`. To avoid such conflicts, the C= header file (*C=.h*) contains a special language customization section that maps C= keywords to their internal codenames, for instance: `#define parallel CPX_KEYWORD_PARALLEL`, which, in effect, enables a programmer to redefine any C= keyword and avoid compilation conflicts.

Another valuable option is the ability to use traditional debugging and profiling instruments in the course of C= program design, development, optimization and maintenance.

See the next section to learn about technology perspectives.

Future Development Avenues

A natural development avenue would be to integrate C= support into C/C++ compilers or implement a source file pre-processing utility for those programmers who value syntactic purity over the flexibility and independence of an extension library.

Another logical move would be to introduce the parallel programming concept of C= in other languages, either sharing C-syntax or not, (for instance, J=, F=, JavaScript=, C#=), as the notion of parallel execution state and task management algorithms implemented in C= are not specific to native compiled languages and can be employed in MRTE/JIT/interpreted languages and programming environments.

The C= concept opens up opportunities of static correctness analysis: since the borders of parallel regions are syntactically defined, it becomes a relatively easy task to detect the illegal use of some constructs within a parallel region. For instance, the use of return operator, long jump, exception handling functions and unary address operators within a parallel or serial scope, where it may be ambiguous or not supported by a certain C= implementation.

As the relationship between parallel scopes and variable visibility scopes is defined, a static correctness analysis tool may hint on the use of the serial operator to synchronize access to certain data, or warn the programmer on possible variable sharing issues, specific to a particular C= implementation. For instance, on the use of local (automatic) variables not through a shared pointer in case C= is not natively supported by a C/C++ compiler.

Dynamic analysis tools may benefit from the fact that the (nested) structure of parallel and serial execution regions is known at any point of program execution. For instance, whenever an unguarded access to a variable is detected, it may be attributed to an appropriate parallel region, and the analysis tool may also suggest inserting a serial operator with the variable's address as the parameter.

C= may as well serve as a unified means of heterogeneous programming, that is, programming for computer systems comprised of central processors and additional parallel co-processor accelerator cards. Consider the following example, whose code maps effectively to a Many-Integrated-Cores or GP-GPU device: the inner parallel region maps to execution units or individual hardware threads, the second-nested region maps to blocks of threads, and the least nested one maps to the high-level cores/devices (if any).

```
void f()
{
    int a;

    parallel()
    {
        int x = pix();
        int b;

        parallel()
        {
            int y = pix();
            int c;

            parallel()
            {
                int z = pix();
                int d;
            }
        }
    }
}
```

The same is true for the data: variable `d` is local to each hardware thread, `c` is stored in a shared thread-block space, `b` may be located at, for instance, the GP-GPU card's global space, while `a` resides in the system memory. Note that `{x,y,z}` variables comprise a hierarchical thread identifier and can be used as data array indices, and the overall logic of parallel data processing is made clearly visible with the use of nested parallel operators.

While not all GP-GPU systems support dynamic generation of parallel work (as implied by the above example) and GPU compilers do not provide for mapping parallel regions to hardware units, the *coload* operator approach may appear to be the easiest means of unifying parallel programming of heterogeneous and distributed systems: a similar principle of automated extraction of portions of data and kernels can be extrapolated to remote systems, which can concurrently copy data of the primary program, transparently execute kernels and copy the results back, mimicking the execution of the *coload* operator by GPUs.

As a result, a single C= program (without modification) may execute on a single-processor, multi-processor, multi-GPU systems and at the same time utilize CPU and GPU resources of remotely connected computers, dynamically balancing its workload and maximizing performance. And again, the semantic principles of C= can be extrapolated to other programming languages, which allows us to speak of the introduction of a Unified Semantic Concept of Parallel Programming after completing the implementation of the above described functionality.

Another development avenue may be in exposing vector instructions of modern processors through the natural programming language semantics: for instance, by allowing an explicit *vector* type modifier to be applied to index variables used in array operations. Thus, accessing an array via such an index may cause the compiler to generate accesses to adjacent elements, as well (as a particular vector register size permits). Assigning said vector elements to other variables will automatically convert those variables to vectors, while referencing other variables (in any non-assignment expression) will make the compiler duplicate the values of the variables being referenced. Such an approach may enable the compiler (and its runtime library) to automatically choose between aligned and unaligned vector memory accesses (by checking the memory alignment at the first reference through a vector index) and will stimulate the introduction of efficient vector scatter-gather operations in new processor architectures. Of course, cases when the control flow depends on values of vector elements will result in serialized execution, but that is very similar to how GP-GPUs handle such cases, and, as GPU practice shows, can be either eliminated or tolerated.

Nearest development plans include porting the current C= library-based solution to other operating systems, subject to the actual interest and demand.

See the next section to get in touch with the authors and find links to the latest information.

Links and Contacts

Latest updates and related information can be found at:

www.hoopoesnest.com

Please address your questions related to sales and licensing to:

sales@hoopoesnest.com

Please send your questions regarding maintenance and support to:

support@hoopoesnest.com

Please forward other related questions and suggestions to:

cstripes@hoopoesnest.com