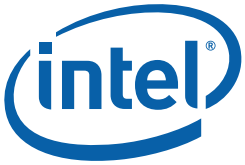

Intel® Concurrent Collections for C++

User's Guide for Version 0.5.x

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Revision Number	Description	Revision Date
1.0	Initial release.	June, 2008
2.0	Update for 0.4 release	September, 2009
3.0	Update for 0.5 release	March, 2010

Contents

§

1 Introduction

Intel® Concurrent Collections for C++ is a language for describing parallel computations. It distinguishes the expression of the potential parallelism of the program from both the lower-level serial details of the computations, and the target-specific details for a particular parallel architecture.

The Intel® Concurrent Collections for C++ parallel programming model allows you to specify the potential parallelism of your application's logic in a declarative manner, separate from your C++ code. Intel® Concurrent Collections for C++ does not provide the main program in your application, and it does not provide the computational logic in your application. You continue to write those parts of your application in C++.

The Intel® Concurrent Collections for C++ language allows you to describe the following elements:

- steps of a computational task
- inputs and outputs of each step
- tags that determine how many times each step will execute

Intel® Concurrent Collections for C++ determines which steps can be run in parallel and executes them. You can use Intel® Concurrent Collections for C++ to control the execution of parallel computations anywhere in your C++ application.

1.1 Overview of Development

You define an Intel® Concurrent Collections for C++ graph which specifies the following:

- *Steps* that define the computations that may be run in parallel. Steps are written in C++ and are invoked by the Intel® Concurrent Collections for C++ run-time library. A step may consume items and can produce items and tags.
- *Items* that define collections of data instances that are inputs or outputs in the graph. Items may not be modified, that is, they are immutable. The same item collection may be input to multiple steps.
- *Tags* that identify instances of steps and items. The same tag collection may be used to control multiple steps.

There are three relationships among the collections in a graph:

- A *consumer* relationship in which items are input to a step.
- A *producer* relationship in which a step creates items and/or tags.
- A *prescription* relationship in which a collection of tags specifies the exact number of times that a step will execute. Every step must be prescribed by a tag collection. Each tag value in the tag collection causes a step to execute once. A step executes when there is both a tag value in its prescribing tag collection, and all of its inputs for that instance of the step are available. The tag value is one of the parameters to a step and is often used to identify the input and output items for the step. New tag values can be added to a tag collection as the graph executes. In this way, the steps in a graph can dynamically control the execution of the graph. Tag values can be any value that you want and have no particular meaning to Intel® Concurrent Collections for C++. As noted above, tag values are often used in identifying inputs and outputs and this often dictates the appropriate values for the tags.

Intel® Concurrent Collections for C++ provides a translator, which is a command line tool that creates a C++ header file from your graph.

You write the code that invokes the graph and the code that implements the steps in C++. Your code includes the header file created by the translator and uses the Intel® Concurrent Collections for C++ run-time library, which controls the parallel execution of your steps.

This User's Guide will use the Intel® Concurrent Collections for C++ sample application primes, in the sections below, to walk through the process of making a serial application use Intel® Concurrent Collections for C++. The steps and procedures in this document assume the samples have been installed in the default location: C:\Program Files\Intel\Concurrent Collections\0.5\Samples on Windows*, and /opt/intel/cnc/0.5/samples on Linux*. If that is not the case, adjust the instructions for your particular installation.

Primes computes the set of prime numbers from 1 to an upper bound specified as an command line argument. The main logic in primes is encapsulated in the following code where the variable n represents the upper bound.

```
//  Allocate an array to hold the discovered prime numbers
int *primes = new int[n];
//  Initial the array with the first prime (2)
int number_of_primes = 0;
primes[ number_of_primes++ ] = 2;
//  From 3 to n, check all odd numbers to determine if they are
//  primes.  This is done by determining the smallest odd number
//  that you can divide the test number by and get a 0 remainder.
//  If the test number is the same as the resulting number, the
//  number is a prime and is added to the array.
for (int number = 3; number <= n; number += 2)
{
   int factor = 3;
   while ( number % factor )
   {
      factor += 2;
   }
   if (factor == number)  // number is prime
```

```
    {
        primes[ number_of_primes++ ] = number;
    }
}
```

1.1.1 Working with the Primes Project

For Linux*, copy the contents of the /opt/intel/cnc/0.5/samples/primes/primes_serial/ directory to an empty work directory. In your work directory, copy primes_serial.cpp to primes.cpp.

The following applies to Windows* only.

To begin, make a copy of the primes_serial sample and modify it for your use; VS200?" is used to represent either "VS2005" or "VS2008".

1. Copy the contents of the C:\Program Files\Intel\Concurrent Collections\0.5\Samples\VS200?\primes\primes_serial directory to an empty work directory, for example, My Documents\Visual Studio 200?\Projects\primes.
2. Open the My Documents\Visual Studio 200?\Projects\primes\primes_serial.sln solution file.
3. Rename the solution to primes by selecting the "Solution 'primes_serial' (1 project)" node in the Solution Explorer. Right click on the node to bring up the context menu and select the Rename entry. Change the name to primes.
4. Rename the project to primes by selecting the "primes_serial" node in the Solution Explorer. Right click on the node to bring up the context menu and select the Rename entry. Change the name to primes.
5. Rename the source file to primes.cpp by expanding the project node and the Source Files folder. Select the "primes_serial.cpp" node in the Solution Explorer. Right click on the node to bring up the context menu and select the Rename entry. Change the name to primes.cpp.
6. Select the "Save All" entry from the Files menu to save the changes.

Continue to use the modified solution throughout the following sections.

1.2 Creating a Graph

There are two ways to represent an Intel® Concurrent Collections for C++ graph: using a graphical syntax and using a textual language syntax. The graphical representation is used at design time only. The text representation is used in the build process by the translator.

Figure 1 (below) shows how collections and relations are represented in the Intel® Concurrent Collections for C++ syntax.

- 2 representations

- 3 types of collections

- 3 types of relationships

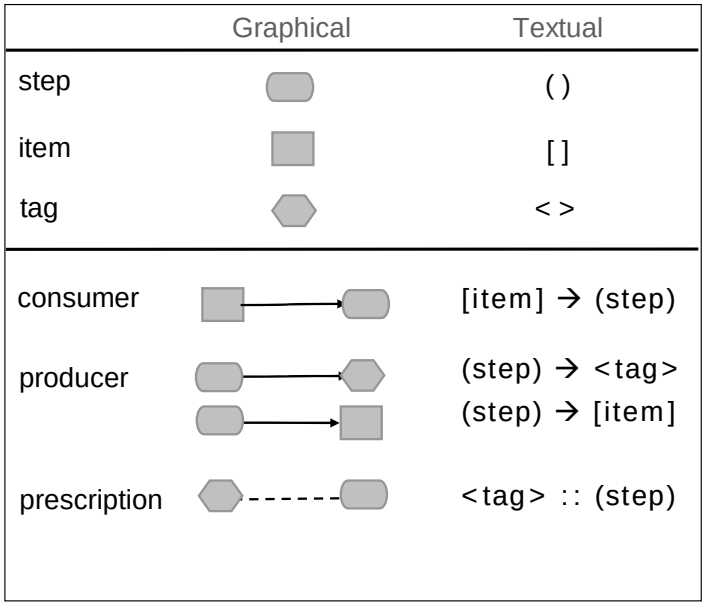

Figure 1: Language Syntax

In addition, cloud shapes, as shown in the figure below, represent the environment, that is your C++ code that invokes the graph.

Figure 2 (below) shows the graphical representation of our parallel primes example.

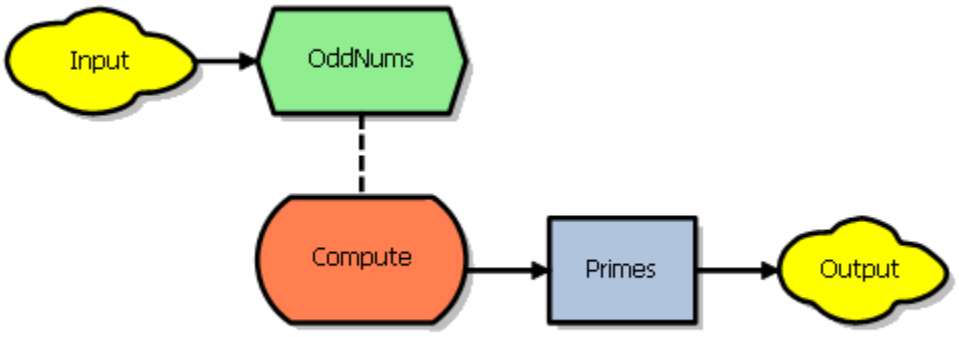

Figure 2: Primes (graphical notation)

You can see that the environment (your C++ code that invokes the graph) initializes the oddNums tag collection. For our primes application the tag values that are put into the oddNums collection are all of the odd numbers from 3 to the upper bound.

The oddNums tag collection prescribes the compute step. This causes the compute step to be invoked for each tag value in the oddNums collection.

The compute step tests the number (the tag value) to determine if it is a prime number. If it is, it adds the number to the primes item collection. Note that the graph illustrates that all of the values in oddNums can be potentially tested in parallel. Since there are no other inputs that are required by the compute step, there are no constraints on the potential parallelism.

When the graph has finished executing, the primes item collection is output to the environment (your C++ code that invokes the graph).

The textual representation corresponding to the graph above is the following:

```
// Declarations

// The tag values are the odd numbers in the range [3..n]
// ("int" is the data type of this tag collection)
<int oddNums>;

// The prime numbers as identified by the compute step
// (The first "int" is the data type of this item collection.
//  The second "int" is the data type of the tag for this item
//  collection)
[int primes <int>];

// Step prescription
// For each oddNums instance, there is compute step to process it
<oddNums> :: (compute);

// Step execution
// The compute step may produce a prime number
(compute) -> [primes];

// Input from the environment: initialize all tags
env -> <oddNums>;

// Output to the environment is the collection of the prime numbers
[primes] -> env;
```

The textual representation is derived from the graphical representation by the following steps:

1. All hexagons (tag collections) on the graph become a tag declaration. Note that in the text, we also specify the C++ data type of the tag.
2. All rectangles (item collections) on the graph become an item declaration. Note that in the text, we also specify the C++ data type of the data item, and the C++ data type of the tag used to identify each item. For example:

   ```
   [int primes <int>];
   ```

3. All dashed lines (prescription relations) on the graph become a prescription relation. For example:
   ```
   <oddNums> :: (compute);
   ```
4. All arrows (consumer or producer relations involving steps) on the graph become a consumer or producer relation. For example:
   ```
   (compute) -> [primes];
   ```
5. All cloud nodes on the graph become env. For example:
   ```
   env -> <oddNums>;
   [primes] -> env;
   ```

See the Textual Notation document for a complete description of the textual syntax.

1.2.1 Working with the Primes Project

For Linux*, copy the text representation of the primes text graph above, and paste it into a new file primes.cnc.

The following applies to Windows* only.

In the primes project:

1. Add a new .cnc file to the project by selecting the “Source File” folder in the Solution Explorer. Right click to bring up the context menu and select the “Add” and then “New Item…” entry.
2. Select the “Utility” category, and then the “Text File (.txt)” icon. In the “Name” field enter primes.cnc. Select “Add”.
3. Double click on primes.cnc to open the new, empty, file in the text editor.
4. Cut the text representation of the primes text graph above, and paste it into primes.cnc.
5. Select the “Save All” entry from the Files menu to save the changes.

1.3 Setting up a Microsoft Visual Studio* Project (for Windows* only)

Intel® Concurrent Collections for C++ provides an Add-in to Microsoft Visual Studio* 2005 and Microsoft Visual Studio* 2008 that allows you to add Intel® Concurrent Collections for C++ settings to any of your projects. To use the Add-in, do the following:

1. Open your project in Microsoft Visual Studio.
2. Select the project node in the Solution Explorer.
3. Select the “Intel® Concurrent Collections” menu entry from the Project menu.
4. Select “Add Concurrent Selection Settings” to add the settings to the current configuration or “Remove Concurrent Selection Settings” to remove the settings. You must add the settings to each configuration separately.

When you add Intel® Concurrent Collections for C++ settings to your project, the Add-in does the following:

1. Adds a property sheet to the active configuration, which:
   a. adds the Intel® Threading Building Blocks and Intel® Concurrent Collections for C++ include paths to the “Additional Include Directories” property of the C++ compiler.
   b. adds the Intel® Threading Building Blocks and Intel® Concurrent Collections for C++ library file paths to the “Additional Library Directories” property of the Linker.
   c. adds the appropriate Intel® Threading Building Blocks and Intel® Concurrent Collections for C++ library files names to the “Additional Dependencies” property of the Linker.
2. Adds the cnc.rules files to the project in order to inform the project build system about which tool needs to be run to process .cnc files (the translator).

Figure 3 (below) shows how your project configuration is set up to build with Intel® Concurrent Collections for C++. You provide one or more text language graphs (.cnc files) and the C++ source files.

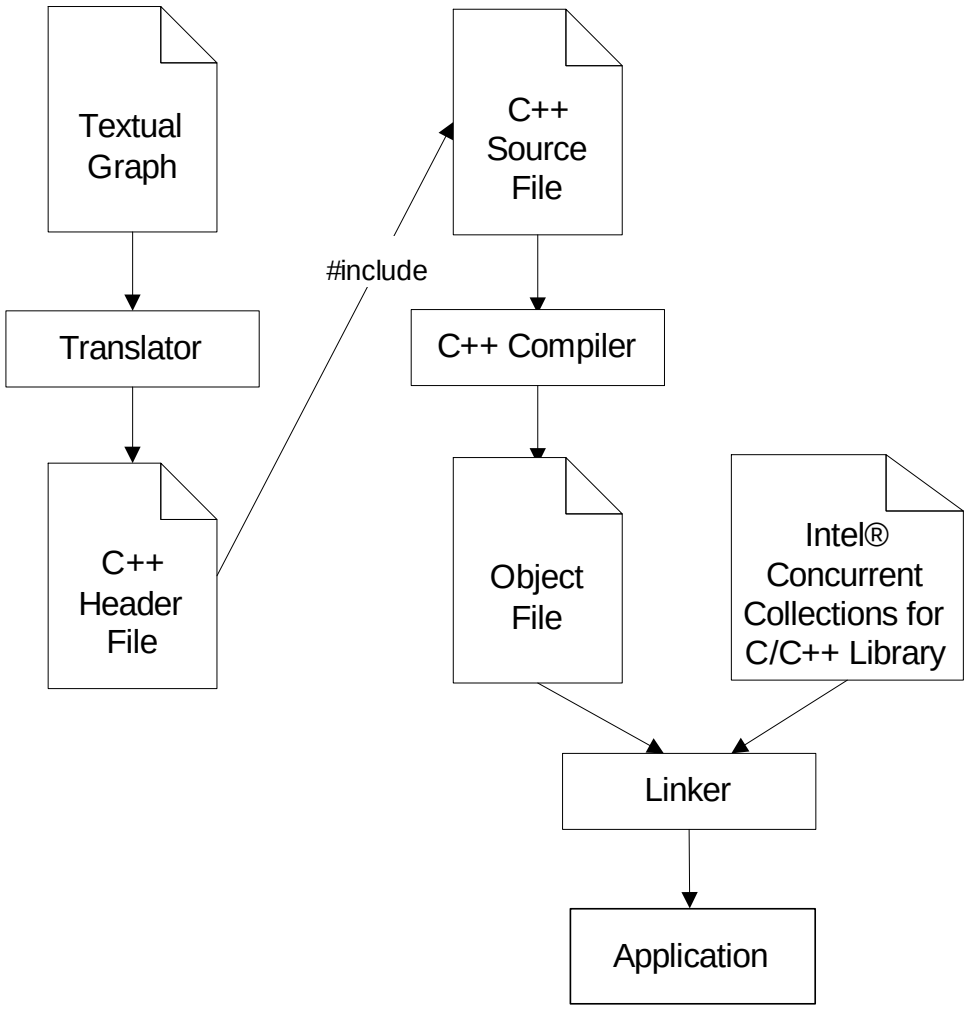

Figure 3: The Build Model

1.3.1 Working with the Primes Project (for Windows* only)

Use the Add-in to add Intel® Concurrent Collections for C++ settings to the primes project we are creating in the tutorial by following the steps shown above.

Do this for both the Debug and Release configurations.

To see where the Add-in modified the project:

- Select the “Property Manager” menu entry from the View menu. Expand the primes node and the debug and release nodes.

The first property sheet in each list should be the “CNC” property sheet.

- In the Solution Explorer, select the “primes” project node. Right click on the project node to bring up the context menu and select the ”Custom Build Rules…” entry.

  You can see that the Concurrent Collections rules file is selected to handle files with the .cnc file extension.

Select the “Save All” entry from the Files menu to save the changes.

1.4 Using the Translator

The translator creates two files from your graph (file-name.cnc):

- A C++ header file named file-name.h
- A text file named file-name_codinghints.txt

The header file contains the definitions of classes derived from the Intel® Concurrent Collections for C++ run-time library classes. The primary class is the context class that represents the graph. It is named file-name_context. It contains instances of classes that represent the collections defined in the graph, and the relationships among the collections. It also contains declarations of step classes and methods that are used in the invocation of your C++ step code.

You should never modify the generated header file. The next time that you run the translator, any changes that you have made will be lost. You don’t need to fully understand all of the details of the header file. Primarily, you only need to know the names of the generated classes – for example, the derived graph class and the collection classes. These names are used in the C++ code that you write.

The “hints” file (file-name_codinghints.txt) contains outlines of the C++ code that you need to write to invoke the graph and implement the steps. We will examine the “hints” file as we implement the code for our primes project.

1.5 Overview of the Run-time Library

The Intel® Concurrent Collections for C++ run-time library is built on top of the Intel® Threading Building Blocks (Intel® TBB). You must use the version of Intel® TBB specified in the software prerequisites; otherwise, linker or run-time errors will occur.

The library performs the bookkeeping to determine when a step is ready to run – that is, there is a tag value in the tag collection that prescribes the step and all of the step’s inputs are available. The library also determines when a graph has terminated. See the “Understanding the Execution Model” section for details.

The library defines a small set of Intel® Concurrent Collections for C++ specific classes. Below is a list of some of the more important classes:

Classes	Description
`context`	Instantiation of a graph; single, derived class per graph
`item_collection`	Collection of items; derived class per item collection
`tag_collection`	Collection of tags; derived class per tag collection

1.5.1 Frequently Used Member Functions

The following member functions are often used:

Function	Description
`context::wait`	Wait for the completion of all steps. Used in all Intel® Concurrent Collections for C++ programs.
`item_collection::get`	Retrieves an item from an item collection. It takes a tag value as its argument. The tag value is used as a "key" into the collection. Item collections are a specialization of a "map" collection, where the tag value is the key and your data item is the mapped value.
`item_collection::put`	Adds an item to an item collection. It takes a tag value (the "key") and your data item as its arguments. item_collection also provides support for iterating through the collection using the item_collection::const_iterator.
`tag_collection_t::put`	Adds a tag to a tag collection and instantiates prescribed steps. It takes a tag value as its argument.

See the Runtime API documentation for a complete description of the public APIs.

1.6 Invoking the Graph

Your C++ code must invoke the graph defined in the generated header file. The basic steps are:

Declare an instance of the derived context (graph) class.

1. Add the initial items and tags to the proper item and tag collections. Specifically, these are the inputs specified by the "env ->" statements in the graph. As soon as tags are added prescribed steps may be started.
2. Wait for completion of all steps by executing the wait() member function.
3. Retrieve any output items and tags that are returned. Specifically, these are the outputs specified by the "-> env" statements in the graph.

1.6.1 Working with the Primes Project

For Linux*, make sure that the translator executable is in your path. If you used the default Intel® Concurrent Collections for C++ installation options, the translator will be located at /opt/intel/cnc/0.5/bin/$ARCH/cnc, where $ARCH is ia32 or intel64. Run the translator on the primes.cnc file with the command:

```
cnc primes.cnc
```

For Windows*, we will now do this for our primes project.

1. Compile the primes.cnc file. Select the file in the Solution Explorer, right click on the node to bring up the context menu, and select the “Compile” entry. Your Output window, Build pane, should show the results of the compilation.
2. Select the “Header Files” folder, right click on the node to bring up the context menu, and select the “Add” and then the “Existing Item..” entry. Select the primes.h file and select “Add”.
3. Select the “primes” project node, right click on the node to bring up the context menu, and select the “Add” and then the “Existing Item..” entry. Change the “Files of type” entry to “*.*”. Select the primes_codinghints.txt file and select “Add”.

Open the primes_codinghints.txt file in the text editor. At the top of the file is a code “outline” of how to invoke the graph. We will cut, paste, and then modify this code in our primes.cpp source file.

1.6.2 Edit 1

Select the code in primes_codinghints.txt between the first and second comments and copy it.

Open the primes.cpp file in the text editor. Navigate to before the line:

```
// Array to hold the primes computed
```

and paste the code that you copied. You can see that the “outline” code corresponds to the basic steps for invoking the graph as defined above. However, the code will require some editing to fit into the serial code we are modifying.

1.6.3 Edit 2

Move the line:

```
#include "primes.h"
```

to the top of the file after the other `#include` lines.

1.6.4 Edit 3

Replace the line:

```
c.oddNums.put(oddNums_Tag);
```

with a loop that adds all of the odd number between 3 and the upper bound to the oddNums tag collection:

```
for (int oddNumber = 3; oddNumber <= n; oddNumber += 2) {
   c.oddNums.put(oddNumber);
}
```

"c" is the name of the instance of the context (graph) class. "oddNums" is the name of the tag collection instance which is member of the context (graph) class. Every item and tag collection has the same member name as the collectons defined in the graph. "put" adds a new tag to the collection. This loop prescribes the number of times that the compute step will run – that is, for each number put into the collection.

1.6.5 Edit 4

Move both the loop inserted in Edit 3 and the following lines:

```
// Wait for all steps to finish
c.wait();
```

To after the line:

```
tbb::tick_count t0 = tbb::tick_count::now();
```

1.6.6 Edit 5

Delete the lines:

```
// For each output to the environment (ENV), get the item
// using the proper tag
int primes_ENV;
```

```
primes.get(...);
```

Insert the printing loop after the line:

```
// Print the prime numbers
```

The printing loop retrieves the prime numbers from the primes item collection. This is done by iterating through the collection. Replace the body of the `if (verbose) { }` block with:

```
// Print the prime numbers
printf("%ld\n", 2);  //  The first prime number (2)

CnC::item_collection<int,int>::const_iterator cii;
for (cii = c.primes.begin(); cii != c.primes.end(); cii++)
{
    int p = *(cii->second);
    printf("%ld\n",p);
}
```

1.6.7 Edit 6

Add the following line before the `printf` statement that prints the number of primes found. It gets the count of prime numbers from the size of the primes collection.

```
number_of_primes = (int)c.primes.size() + 1;
```

1.6.8 Edit 7

Now we will delete some code from the serial version that is no longer necessary. Since we no longer need an array to hold the prime numbers (they are now put into the primes item collection) delete the lines:

```
// Array to hold the primes computed
int *primes = new int[n];
```

and delete the line that initializes the array:

```
primes[ number_of_primes++ ] = 2;
```

1.6.9 Edits Complete

Select the “Save All” entry from the Files menu to save the changes.

We have completed the code to invoke the graph. In the next section we will create the code for the compute step.

1.7 Implementing a Step

Every step has the same function signature. For example:

int compute::execute(const int & t, FindPrimes_context & c) const

Every step function returns a value from the StepReturnValue_t enumeration. Currently, the value CNC_Success should always be returned.

Every step function takes two arguments:

1. The tag value for the current invocation of the step. The tag value is often used when getting items from the collections and/or putting items and tags into the collections.
2. The instance of the context (graph) class that is invoking the step. The context instance is used to access the graph collections from the step.

Every step does the following:

1. (Optionally) Gets its input items from the graph’s collections. It is possible that a step requires no additional item other than its tag value. This is true in our example where the number to test is used as the tag value.
2. Performs its computations.
3. (Optionally) Puts output items into the graph’s collections. It is often the case that output items are only put under certain conditions. This is true in our example where the number is added to the primes item collection only when it is determined that it is a prime number.

1.7.1 Working with the Primes Project

We will now create the compute step for our primes project.

1.7.2 Edit 1

Select the code in primes_codinghints.txt after the second comment and copy it. Paste the code that you copied to immediately after the `#include` statements.

1.7.3 Edit 2

The “outline” step code has a place to put the step computations. Move the following lines that we left in the main function in the previous section:

```
for (int number = 3; number <= n; number += 2)
{
    int factor = 3;
    while ( number % factor )
    {
        factor += 2;
    }
    if (factor == number)  // number is prime
    {
        primes[ number_of_primes++ ] = number;
    }
}
```

and replace the lines:

```
// Step implementation logic goes here
...
```

In our step, we are not looping through all of the possible prime numbers as we did in the serial example. Instead each step is testing one prime number. This gives us the potential to execute the steps in parallel.

We need to remove the loop and use the tag value passed into the step. Remove the lines:

```
for (int number = 3; number <= n; number += 2)
{
```

and the terminating `}`.

Add the following line to the beginning of the step:

```
int number = t;
```

1.7.4 Edit 3

Our final change is to indicate where to put a prime number when our step determines that it has found one. In the serial code, the prime number was added to the primes array. In this version, the prime number is added to the primes item collection. Replace the line:

```
primes[ number_of_primes++ ] = number;
```

With the finished version of the “outline” put method invocation::

```
c.primes.put(t, number);
```

Delete the explanatory comments that were part of the “skeleton” code.

1.7.5 Edits Complete

Select the “Save All” entry from the Files menu to save the changes.

1.8 Running the Sample Application

After you have built your application, you can run and debug it as with any other C++ application. (For Linux*, see the “Getting Started” document for important setup instructions.) Some suggestions for debugging are provided below. For Linux*, make sure that you have invoked the appropriate shell script included with Intel® TBB in order to set up the relevant environment variables. For Windows*, make sure that the Intel® TBB dll directory is part of your system Path environment variable.

1.8.1 Working with the Primes Project

primes accepts the following command arguments:

```
primes [-v] n
```

The –v switch is optional and produces “verbose” output. `n` is required and specifies the upper bound of the numbers to check.

For Windows*, set the command arguments for invoking the primes application in the project properties. Select the “primes” project node in the Solution Explorer. Right click on the node to bring up the context menu. and select the “Properties” entry. Select the Configuration Properties, Debugging category and enter “-v 25” into the “Command Arguments” property.

Run the primes project. For Linux*, you can simply specify “primes –v 25” on the command line. For Windows*, this is done by selecting Ctrl + F5 (run without debugging). The output from the primes application should look similar to the following example:

```
Determining primes from 1-25
Found 9 primes in 9.49841e-005 seconds
2
3
5
7
11
```


```
13
17
19
23
Press any key to continue . . .
```

1.9 Understanding the Execution Model

Intel® Concurrent Collections for C++ has a simple model that every computation step specified in the graph is assumed able to be executed in parallel with the only constraints imposed by the producer and consumer relations in the graph.

As the program executes, item, tag and step instances monotonically accumulate attributes indicating their state.

- If a step or the environment produces an item, then the item becomes available.
- If a step or the environment produces a tag, then the tag becomes available.
- If a tag is available and its tag collection prescribes a step, then the step becomes prescribed for that tag.
- If all of the inputs of a step are available, then the step is inputs-available.
- If a step is both inputs-available and prescribed, then it is enabled.

Any enabled step is ready to execute. Once the step has completed executing, then it is executed. No order of execution is implied by any interpretation of “order” in the tag collection. For example, steps are not executed in the order that tags are added to a tag collection.

When an item or tag is determined to have no further uses, then it is dead.

The *execution frontier*, the set of instances that are of any interest at some particular time, evolves during execution. This is that set of instances that have any attribute but are not yet dead (for items and tags) or executed (for steps).

Program termination occurs when no step is currently executing and no unexecuted step is currently enabled.

Valid program termination occurs when a program terminates and all prescribed steps have executed.

1.10 Debugging your application

For Windows*, the typical Microsoft Visual Studio project has Debug and Release configurations. When you use the Intel® Concurrent Collections for C++ Add-in to add settings to your configuration, if the name of the configuration contains the string “Debug”, the Add-in will set up the configuration to link with the debug versions of the Intel® Concurrent Collections for C++ and Intel® TBB run-time libraries. Otherwise, it will set up the configuration to use the non-debug versions of the libraries.

You debug your application as you would any other C++ application. From the time that step tags are put to enable step executions until the time that the graph terminates, your steps will be called in a non-deterministic order by the run-time library. There may also be multiple instances of the same or different steps running at the same time. You cannot directly "step into" your step code, and so settings breakpoints in your step code is the usual technique.

The current implementation of the Intel® Concurrent Collections for C++ run-time library has a behavior that you need to be aware of when debugging your step code. A step may begin execution before its input items are available. If an item is unavailable, an exception will be thrown. The runtime catches the exception and re-starts the step once the item became available.

To write steps safely, your step code should:

- always get all items before putting tags or items
- defer allocation and computation to after last get

The above rules protect you from memory leaks, dead-locks, and inefficient code.

It also means that you may encounter the following behavior when stepping through your step code. If you set a breakpoint before all of the gets have been done, and step through your code, as you step over a Get member function call, you may encounter cases where the next statement is not reached. This is because an exception has been thrown and the step has exited and has been re-scheduled for later execution by the run-time library.

1.10.1 Using Tracing utility

Intel® Concurrent Collections for C++ provides a convenient utility for tracing program execution. In the coding hints file there is a commented line:

```
//    CnC::debug::trace( c.oddNums, "oddNums" );
```

If uncommented this line enables tracing utility for the tag collection, it means that all actions performed on this collection will be logged in a human-readable format.

Similar tracing utility can be applied to item collections.

1.11 Using the Samples

The Intel® Concurrent Collections for C++ kit provides samples. For Windows* the samples can be used with Microsoft Visual Studio* 2005 or Microsoft Visual Studio* 2008. The appropriate version(s) of the samples are installed based upon the version(s) of Visual Studio you have installed on your system.

See the readme.txt supplied with each sample for more details about each sample.

1.12 Deploying Programs that use Intel® Concurrent Collections for C++ (for Windows* only)

The Intel® Concurrent Collections for C++ run-time library is a small static library that is linked with your application. It does, however, use Visual C++ run-time libraries, which are always distributed as dlls, and the Intel® TBB run-time libraries, which are typically distributed as dlls.

In order to run an Intel® Concurrent Collections for C++ application on a system without Visual C++ or Intel® TBB installed, you must redistribute the proper dlls. See the Visual C++ "Deployment" topic in the Visual C++ on-line documentation for the various ways to redistribute Visual C++ dlls. See the redist.txt file in your Intel® TBB installation for the list of dlls that you can redistribute.

Your application will typically use the following Intel® TBB dlls:

Configuration	Libraries
Debug	tbb_debug.dll tbbmalloc_debug.dll
Release	tbb.dll tbbmalloc.dll

You must ensure that these dlls are placed where the Windows* operating system can find them when it loads your application. Two common methods are to place the dlls in the same directory as the application or to add the directory that contains the dlls to your system's Path environment variable.