# Intel(R) Threading Building Blocks

## Getting Started Guide

Intel® Threading Building Blocks is a runtime-based parallel programming model for C++ code that uses threads. It consists of a template-based runtime library to help you harness the latent performance of multicore processors. Use Intel® Threading Building Blocks to write scalable applications that:

- Specify logical parallel structure instead of threads
- Emphasize data parallel programming
- Take advantage of concurrent collections and parallel algorithms

This guide provides a complete example that uses Intel® Threading Building Blocks to write, compile, link, and run a parallel application. The example shows you how to explore a key feature of the library and to successfully build and link an application. After completing this guide, you should be ready to write and build your own code using Intel® Threading Building Blocks.

## Contents

# 1 Note Default Directory Paths

Before you begin, make sure you have successfully installed Intel® Threading Building Blocks on your machine. Otherwise, install it according to the instructions in `INSTALL.txt`.

The default installation locations for the `bin`, `doc`, and `examples` directories used in this document are shown in the following table:

| Platform | Default Directories |
|---|---|
| Linux* OS | `/opt/intel/composer_xe_2011_sp1/tbb/[bin|examples]`<br>`/opt/intel/composer_xe_2011_sp1/Documentation` |
| Mac OS* X | `/opt/intel/composer_xe_2011_sp1/tbb/[bin|examples]`<br>`/opt/intel/composer_xe_2011_sp1/Documentation` |
| Windows* OS with Intel® IA-32 processor | `C:\Program Files\Intel\ComposerXE-2011 SP1\tbb\examples`<br>`C:\Program Files\Intel\ComposerXE-2011 SP1\Documentation` |
| Windows* OS with Intel® 64 Instruction Set Architecture (ISA) processor | `C:\Program Files (x86)\Intel\ComposerXE-2011 SP1\tbb\examples`<br>`C:\Program Files (x86)\Intel\ComposerXE-2011 SP1\Documentation` |

# 2 Set Up Environment

Before using Intel® Threading Building Blocks, you must register the environment variables that are used to locate necessary library and include files as follows:

1. Locate the configuration scripts for your operating system. The scripts are located in the `bin` directory.
2. Execute the appropriate scripts or set properties for your operating system:
   - On Linux* and Mac OS* X operating systems, from the `bin` directory, source the `tbbvars.[c]sh` script. These scripts modify the paths held by the `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` (Mac OS* X), and `CPATH` variables, and affect only your current shell.
   - On Windows* operating systems, modify your Visual Studio project build configurations (debug, release) properties as follows:

     **C/C++** Properties

     - **General**: add an additional include directory:
       "$(TBBROOT)\include"

     **Linker** Properties

     - **General**: add an additional library directory (shown for Visual Studio 2005 32-bit library):
       `$(TBBROOT)\lib\ia32\vc8`
     - **Input**: add an additional dependency
       `tbb_debug.lib` or `tbb.lib`

# 3 Develop an Application Using parallel_for

This section presents a basic example that uses the `parallel_for` template in a substring matching program. For each position in a string, the program displays the length and location of the largest matching substring elsewhere in the string.

Consider the string "`babba`" as an example. Starting at position `0`, "`ba`" is the largest substring with a match elsewhere in the string (position `3`).

You can refer to a complete version of this program in the Intel® Threading Building Blocks (Intel® TBB) `examples/GettingStarted` folder. Or you can follow the step-by-step development of this application given here. In this section, new code that is added in each step is shown in **blue**. Code that is carried over from a previous step is shown in black. Lines are numbered in the order they appear in the final completed example.

**To develop the example code:**

— Create a new empty application.

— The `using` statement imports the namespace `tbb`, in which all of the library's classes and functions are found (line 07).

```
07:    using namespace tbb;

36:    int main() {
50:      return 0;
51:    }
```

3. Create the example string that is transformed by the program (lines 38 - 40) and the arrays for holding the lengths of the largest matched substrings and their locations (lines 42 - 43).
   The example generates a Fibonacci string consisting of a series of '`a`' and '`b`' characters.
4. Add statements to output the lengths and locations of the largest substring matches for each position (lines 46 - 47).

```
01:      #include <iostream>
02:      #include <string>

07:      using namespace tbb;
08:      using namespace std;
09:      static const size_t N = 23;
```

```
36:      int main() {

38:        string str[N] = { string("a"), string("b") };
39:        for (size_t i = 2; i < N; ++i) str[i] = str[i-1]+str[i-2];
40:        string &to_scan = str[N-1];
41:        size_t num_elem = to_scan.size();

42:        size_t *max = new size_t[num_elem];
43:        size_t *pos = new size_t[num_elem];

44—45:     // will add code to populate max and pos here

46:        for (size_t i = 0; i < num_elem; ++i)
47:          cout << " " << max[i] << "(" << pos[i] << ")" << endl;
48:        delete[] pos;
49:        delete[] max;
50:        return 0;
51:      }
```

5. Add a call to the `parallel_for` template function (lines 44 - 45).
   The first parameter of the call is a `blocked_range` object that describes the iteration space.
   `blocked_range` is a template class provided by the Intel® Threading Building Blocks library. The constructor takes three parameters:
   — The lower bound of the range.
   — The upper bound of the range.

   The second parameter to the `parallel_for` function is the function object to be applied to each subrange.

```
01:    #include <iostream>
02:    #include <string>
05:    #include "tbb/parallel_for.h"
06:    #include "tbb/blocked_range.h"

07:    using namespace tbb;
08:    using namespace std;
09:    static const size_t N = 23;

36:    int main() {

38:      string str[N] = { string("a"), string("b") };
39:      for (size_t i = 2; i < N; ++i) str[i] = str[i-1]+str[i-2];
40:      string &to_scan = str[N-1];

41:      size_t num_elem = to_scan.size();
```

```
42:       size_t *max = new size_t[num_elem];
43:       size_t *pos = new size_t[num_elem];


44:       parallel_for(blocked_range<size_t>(0, num_elem ),
45:                    SubStringFinder( to_scan, max, pos ) );

46:       for (size_t i = 0; i < num_elem; ++i)
47:         cout << " " << max[i] << "(" << pos[i] << ")" << endl;
48:       delete[] pos;
49:       delete[] max;
50:       return 0;
51:     }
```

6. Implement the body of the `parallel_for` loop (lines 10 – 35).
   At runtime, the template `parallel_for` automatically divides the range into subranges and invokes the `SubStringFinder` function object on each subrange.
7. Define the class `SubStringFinder` (line 10) to populate the max and pos array elements found within the given subrange.
   At line 16, the call `r.begin()` returns the start of the subrange and the `r.end()` method returns the end of the subrange.

```
01:     #include <iostream>
02:     #include <string>
03:     #include <algorithm>
04:     #include "tbb/parallel_for.h"
05:     #include "tbb/blocked_range.h"

07:     using namespace tbb;
08:     using namespace std;
09:     static const size_t N = 23;

10:     class SubStringFinder {
11:       const string str;
12:       size_t *max_array;
13:       size_t *pos_array;
14:     public:
15:       void operator() ( const blocked_range<size_t>& r ) const {
16:         for ( size_t i = r.begin(); i != r.end(); ++i ) {
17:           size_t max_size = 0, max_pos = 0;
18:           for (size_t j = 0; j < str.size(); ++j)
19:           if (j != i) {
20:             size_t limit = str.size()-max(i,j);
21:             for (size_t k = 0; k < limit; ++k) {
22:               if (str[i + k] != str[j + k]) break;
23:               if (k > max_size) {
24:                 max_size = k;
25:                 max_pos = j;
```

```
              }
            }
          }
          max_array[i] = max_size;
          pos_array[i] = max_pos;
        }
      }
      SubStringFinder(string &s, size_t *m, size_t *p) :
        str(s), max_array(m), pos_array(p) { }
    };

    // The function main starting at line 36 goes here
```

# 4 Build the Application

Intel® Threading Building Blocks is compatible with the GCC* and Microsoft compilers. This section assumes that you are using the Intel® C++ Compiler. You can use the GCC or Microsoft C++ compilers interchangeably in the directions given below.

## Building Code from the Examples Directory

If you did not type the example in **Develop an Application Using `parallel_for`**, build from the completed source code provided in the `examples/GettingStarted` folder.

**Linux* or Mac OS* X Systems**

1. `cd` to the directory `examples/GettingStarted/sub_string_finder/`.
2. Type `make` to build and run the example.

**Windows* Systems**

1. Invoke Visual Studio on the file `examples\GettingStarted\sub_string_finder\msvs\sub_string_finder.sln` using one of the following methods:
   — Browse to the directory containing `sub_string_finder.sln` and double-click the file.
   — Invoke Visual Studio from the Start menu, and then open the `sub_string_finder.sln` file via **File → Open → Open Project**.
2. Press <Ctrl-F5> to build and run the example.

## Building Manually Typed Code

If you manually typed the code provided in **Develop an Application Using** `parallel_for`, build your application by invoking the appropriate compiler directly:

**Linux* or Mac OS* X Systems**

Use the command line:

```
icc sub_string_finder.cpp -ltbb
```

**Windows* Systems from the Command Line**

From within the Intel® C++ Compiler build environment issue the following command:

```
icl /MD sub_string_finder.cpp tbb.lib
```

# 5 Run the Application

To run the application you built:

1. Run the application as you would normally. When run, the program outputs a long list of length and location pairs.
2. Optionally, to compare the performance of this example to a sequential version, you can build and run the extended version of the `SubStringFinder` example located in the Intel® Threading Buildings Blocks `examples/GettingStarted` folder as `sub_string_finder_extended.cpp`. This extended example calculates and displays the speedup obtained by using Intel® Threading Building Blocks to parallelize the algorithm compared to performing the same work sequentially.

# 6 Next Steps

To get the most out of the Intel® Threading Building Blocks library, explore the following additional resources.

1. **Tutorial** is a document that walks you through the major classes, algorithms and concepts used by Intel® Threading Building Blocks. This document is available in the `doc` directory as `Tutorial.pdf`.
2. **Design Patterns** is a "cookbook" of some common parallel programming patterns using Intel® Threading Building Blocks.
3. **Reference** is a complete, detailed reference manual for all the functions and interfaces provided by Intel® Threading Building Blocks. It is available in the `doc` directory as `Reference.pdf`.
4. **Examples** includes a collection of example programs that demonstrate the various features of Intel® Threading Building Blocks. These programs are located within the `examples` directory. A good place to start is with the extended version of the `SubStringFinder` example presented in this **Getting Started Guide.** This extended example is found in the `examples/GettingStarted/sub_string_finder` subdirectory as `sub_string_finder_extended.cpp`. When run, it calculates and displays the speedup obtained by using Intel® Threading Building Blocks to parallelize the algorithm compared to performing the same work sequentially.
5. **Doxygen** includes documentation that was automatically generated from the comments in the Intel® Threading Building Blocks include files. The `Doxygen` subdirectory is found within the `doc` directory. The files in the `Doxygen` directory are in HTML format and are viewable with any browser that supports HTML.

# Legal Information

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

## Revision History

| Document Number | Revision Number | Description | Revision Date |
|---|---|---|---|
| 314904-009 | 009 | Updated the Copyright Date | April 2012 |
| 314904-008 | 008 | Updated the Optimization Notice | October 2011 |
| 314904-007 | 007 | Updated install paths. | August 2011 |
| 314904-006 | 006 | Modified to match source code in example. | October 2010 |
| 314904-005 | 005 | Correct paths for library version 3.0. | June 2010 |