



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 tasks 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

| | |
|---|----|
| Disclaimer and Legal Information | 2 |
| 1 Note Default Directory Paths..... | 3 |
| 2 Register Environment Variables (Linux* and Mac OS* X only) | 4 |
| 3 Develop an Application Using parallel_for | 5 |
| 4 Build the Application..... | 9 |
| 5 Run the Application | 11 |
| 6 Next Steps..... | 12 |



Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL(R) PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting [Intel's Web Site](http://www.intel.com).

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright (C) 2005 - 2008, Intel Corporation. All rights reserved.

Revision History

| Document Number | Revision Number | Description | Revision Date |
|-----------------|-----------------|---------------------------|---------------|
| 314904-001 | 001 | Applied new template. | August 2006 |
| | 002 | Updated legal disclaimer. | May 2008 |



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

| Platform | Default Directories |
|--|--|
| Linux* | /opt/intel/tbb/<version>/ [bin doc examples] |
| Mac OS* X | /Library/Frameworks/Intel_TBB.framework/Versions/<version>/ [bin doc examples] |
| Windows* with Intel® IA-32 processor | C:\Program Files\Intel\TBB\<version>\ [doc examples] |
| Windows* with Intel® 64 Instruction Set Architecture (ISA) | C:\Program Files (x86)\Intel\TBB\<version>\ [doc examples] |



2 *Register Environment Variables (Linux* and Mac OS* X only)*

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 for your operating system. On Linux* and Mac OS* X 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.



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 `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:

1. Create a new empty application.
In the main function, initialize the task scheduler by instantiating a `task_scheduler_init` object (line 37).
 - Any thread that uses an Intel® Threading Building Blocks algorithm must have an initialized `task_scheduler_init` object. In this example, the default constructor for the `task_scheduler_init` object informs the task scheduler that the thread is participating in task execution, and the destructor informs the scheduler that the thread no longer needs the scheduler.
 - The definition for the `task_scheduler_init` class is included from the `task_scheduler_init.h` header file (line 04).
 - The `using` statement imports the namespace `tbb`, in which all of the library’s classes and functions are found (line 07).

```
04:  #include "tbb/task_scheduler_init.h"

07:  using namespace tbb;

36:  int main() {
37:      task_scheduler_init init;
47:      return 0;
48:  }
```

2. 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 41 - 42).



The example generates a Fibonacci string consisting of a series of 'a' and 'b' characters.

3. Add statements to output the lengths and locations of the largest substring matches for each position (lines 45 - 46).

```
01:      #include <iostream>
02:      #include <string>
04:      #include "tbb/task_scheduler_init.h"

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

36:      int main() {
37:          task_scheduler_init init;

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 *max = new size_t[to_scan.size()];
42:          size_t *pos = new size_t[to_scan.size()];

43-44:      // will add code to populate max and pos here

45:          for (size_t i = 0; i < to_scan.size(); ++i)
46:              cout << " " << max[i] << "(" << pos[i] << ")" << endl;
47:          return 0;
48:      }
```

4. Add a call to the `parallel_for` template function (lines 43 - 44).
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 `<grainsize>`. The `parallel_for` subdivides the range into sub-ranges that have approximately `<grainsize>` elements.

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

```
01:      #include <iostream>
02:      #include <string>
04:      #include "tbb/task_scheduler_init.h"
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() {
37:      task_scheduler_init init;

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 *max = new size_t[to_scan.size()];
42:      size_t *pos = new size_t[to_scan.size()];

43:      parallel_for(blocked_range<size_t>(0, to_scan.size(), 100),
44:                  SubStringFinder( to_scan, max, pos ) );

45:      for (size_t I = 0; I < to_scan.size(); ++i)
46:          cout << " " << max[i] << "(" << pos[i] << ")" << endl;
47:      return 0;
48:  }

```

5. 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.
6. 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/task_scheduler_init.h"
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;

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;
26:                     }
27:                 }
28:             }
29:         max_array[i] = max_size;
30:         pos_array[i] = max_pos;
31:     }
32: }
33: SubStringFinder(string &s, size_t *m, size_t *p) :
34:     str(s), max_array(m), pos_array(p) { }
35: };

36-48: // 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 `examples/GettingStarted/sub_string_finder` directory.
2. Type “make” to build and run the example.

Windows* Systems

NOTE: This section uses Visual Studio* .NET* 2003 (vc7.1\) but you can use Visual Studio 2005 by replacing `vc7.1\` with `vc8\`.

1. Invoke Visual Studio on the `examples\GettingStarted\sub_string_finder\vc7.1\sub_string_finder.sln` file 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 Building 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. **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`.
3. **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.
4. **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.