Intel® Parallel Building Blocks: Getting Started Tutorial and Hands-on Lab
Introduction

Nomenclature

- Intel® Parallel Building Blocks—Intel® PBB
- Intel® Array Building Blocks—Intel® ArBB
- Intel® Threading Building Blocks—Intel® TBB
- Intel® Cilk™ Plus

Requirements for this evaluation guide:

- Microsoft Visual Studio 2008* or 2010*

As microprocessors transition from Ghz as the primary reason for performance gains to features such as multiple cores, it is increasingly important for developers to optimize their applications to take advantage of these new platform capabilities. In the past, the same application would automatically perform better on a newer CPU with higher clock speeds. However, when customers buy computers with the latest CPUs, they may not see a corresponding increase in applications that are written in serial (or sequential). In order to offer the increase in performance users expect to see, developers need to understand the new features available on the CPU and take advantage of them.

The Intel software group provides a range of tools specifically designed to help developers parallelize their applications. In this guide, we’re going to cover Intel® Parallel Building Blocks (Intel® PBB), a comprehensive and complementary set of models that provide solutions ranging from general-purpose to specialized parallelism. The tools within Intel PBB allow developers to mix and match solutions within an application to suit their specific environment and needs, providing a simple yet scalable way to embrace multicore era.

This hands-on evaluation guide contains a wealth of information to get started with Intel PBB:

- A comprehensive explanation/disclosure of each of the models
  - Recommendations on when to use a particular model
- Three videos describing how each of the models works
- Full beginner and advanced Microsoft Visual Studio* Intel PBB lab coding projects set up for a 64-bit system.

![Figure 1](image1.png)
Intel Parallel Building Blocks: An Overview

How can you code at a high level of abstraction while still extracting high levels of parallelism? Naturally, Intel customers already have programs that are actively building, and the Intel PBB models support the wide variety of ways people program for multicore hardware. With Intel PBB, programmers augment the computationally intensive kernels with Intel PBB parallelism abstractions, or they can start from scratch and more rapidly create new parallel applications.

Intel PBB is not the name of a product. Rather, it is a term for a collection of models (or tools) that assist developers with implementing parallelism. It consists of Intel® Cilk™ Plus, Intel TBB, and Intel ArBB.

<table>
<thead>
<tr>
<th>Intel Parallel Building Blocks</th>
<th>Intel Threading Building Blocks</th>
<th>Intel Array Building Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What is it?</strong></td>
<td>Language extensions for task and data parallelism</td>
<td>Widely used C++ template library for task parallelism</td>
</tr>
<tr>
<td><strong>Primary features</strong></td>
<td>• Three simple keywords and array notations for parallelism</td>
<td>• Parallel algorithms and Data structures</td>
</tr>
<tr>
<td></td>
<td>• Support for task, data, and vector code</td>
<td>• Scalable memory allocation and task scheduling</td>
</tr>
<tr>
<td></td>
<td>• Similar execution as serial code</td>
<td>• Synchronization primitives</td>
</tr>
<tr>
<td><strong>Reasons to use</strong></td>
<td>• Simple way to parallelize your code</td>
<td>• Rich feature set for general-purpose parallelism</td>
</tr>
<tr>
<td></td>
<td>• Use when compiler assistance is valued</td>
<td>• C++, Windows, Linux, MacOS, other OSs</td>
</tr>
<tr>
<td></td>
<td>• C and C++ Windows* and Linux*</td>
<td></td>
</tr>
</tbody>
</table>

Since Intel sells "more than hammers," the programmer does not have to force every problem "into a nail." Therefore, the following guide is highly descriptive about the Intel PBB models, but does not necessarily have to be prescriptive. Nearly all problems in parallelism can be expressed with one of these building blocks or a combination. Hence, the greatest strength of Intel PBB is both its diversity of models and its consistent expressiveness.

Intel PBB models are easy-to-use template libraries or language extensions that allow the programmer to gain great performance benefits through parallelism while still maintaining a high degree of productivity. Intel PBB excels in the following four areas:

**Usability**

The PBB models represent general-purpose programming with high levels of abstraction.

**Reliability**

The Intel PBB models are compatible with existing Intel tools, which can help eliminate common coding and threading errors.

**Scalability**

The Intel PBB models support forward scaling to more processor cores. Adopting parallelism through Intel PBB is an investment that continues to provide benefits as the number of CPU cores increases.

**Openness**

The Intel PBB models are portable to multiple operating systems, hardware, integrated development environments (IDE), and compiler platforms with flexible licensing.

There will always be programmers that want to extract the last bits of optimization. Intel PBB permits them to do just that. The models allow experts to dig down closer to the underlying details if they choose, yet still open up parallel programming for all types of programmers.
Considerations for the Parallel Programmer

A parallel programmer today must extract parallelism inside each individual core through SIMD instruction sets in addition to parallelism in-between cores through threading. Parallelism with threads isn’t new, but Posix* and Windows* threads were designed to be part of the operating system, not the language itself. Intel has evolved this model through the abstractions with Intel® Threading Building Blocks (Intel® TBB), the market-leading model for parallelism. Intel TBB programmers work with tasks, not individual threads. The full suite of Intel® Developer Products supports varying degrees of control over vectorization and threading, from beginner to advanced programmer, with Intel® Parallel Composer 2011 as the backbone. Intel Cilk augments Intel TBB for usability and safety guarantees. Intel ArBB uses Intel TBB as its threading runtime while also generating SIMD instruction sets tailored to the architecture in question.

Next, the programmer must see whether their problem can best be solved using a data parallel or general-purpose parallel approach. General-purpose parallelism involves managing tasks to solve one problem combined with a parallel algorithm. Data parallelism is a special case of parallelism. It involves the application of the same code to multiple data items without specifying the order of execution. Implemented using a combination of vector parallelism, loops, and tasks, it is more applicable when data items can be accessed simultaneously, or when access time is negligible relative to the computation. However, data parallelism is not to be confused with vectorization, which can be used in both general-purpose and data parallel approaches.

Intel PBB provides a comprehensive and complementary set of programming constructs, delivering one proven, cohesive, and interoperable solution that covers today’s and tomorrow’s parallel programming needs.

- You can access both data parallel and general-purpose parallelism solutions that work seamlessly with the tools you already use for existing applications.
- You can leverage both language extension and library solutions, strengthening your confidence in your application by using a solution that is proven and deployed in the industry.
- Intel PBB benefits from both optimized high-level algorithms and low-level constructs to build custom workloads. Maximize your performance today and tomorrow through automatic application scaling, while still retaining your desired level of control.
- You can mix and match new parallel models within an application to suit your environment/application, minimizing the need for more coding as platforms evolve.
Intel Threading Building Blocks (Intel TBB)

What is it? What are the benefits?

Intel TBB is a C++ library that provides components for general shared-memory parallelism. It is easy to integrate incrementally into existing environments. Designed for nested parallelism, it also contains high-level generic components. Intel TBB offers template code for common idioms, yet is customizable to accommodate specific use cases and provides low-level components for expert programmers who need more control. It is also general enough to build custom algorithms and arbitrary task graphs. Finally, one can use Intel® Parallel Inspector and Intel® Parallel Amplifier in conjunction with Intel TBB.

When should you use Intel TBB?

Call on Intel TBB when you are interested in general-purpose parallelism, such as when special-purpose constructs of data parallelism don’t apply. Also, use it when its well-developed algorithms and containers are applicable to your: parallel sort, concurrent vectors, etc.

Key Features

<table>
<thead>
<tr>
<th>Generic Parallel Algorithms</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel_for(range)</td>
<td>concurrent_hash_map</td>
</tr>
<tr>
<td>parallel_reduce</td>
<td>concurrent_queue</td>
</tr>
<tr>
<td>parallel_for_each(begin, end)</td>
<td>concurrent_bounded_queue</td>
</tr>
<tr>
<td>parallel_do</td>
<td>concurrent_vector</td>
</tr>
<tr>
<td>parallel_invoke</td>
<td>concurrent_unordered_map</td>
</tr>
<tr>
<td>parallel_pipeline; pipeline</td>
<td></td>
</tr>
<tr>
<td>parallel_sort</td>
<td></td>
</tr>
<tr>
<td>parallel_scan</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task scheduler</th>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_group</td>
<td>atomic</td>
</tr>
<tr>
<td>task_structured_group</td>
<td>mutex; recursive_mutex</td>
</tr>
<tr>
<td>task_scheduler_init</td>
<td>spin_mutex; spin_rwlock_mutex</td>
</tr>
<tr>
<td>task_scheduler_observer</td>
<td>queuing_mutex; queuing_rwlock_mutex</td>
</tr>
<tr>
<td></td>
<td>null_mutex; null_rwlock_mutex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th>Threads</th>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick_count</td>
<td>thread</td>
<td>tbb_allocator; cache_aligned_allocator; scalable_allocator; zero_allocator</td>
</tr>
</tbody>
</table>
**Exercise 1: Sorting and Counting Using Intel® Threading Building Blocks**

**Difficulty Level:** Beginner  
**Lab Project Setting:** Microsoft Visual Studio 2008* (64-bit project)  
**Materials Needed:**  

**Objective:** This lab uses a simple program that manipulates a database of personal information, stored as std::vector (see type People). The program sorts the information by age, and then counts occurrences of each name. In this exercise, you will use Intel TBB to optimize the sorting and memory management.

Why use this PBB? This program uses a central data structure, allocates and deallocates significant amounts of memory, and manipulates vectors in ways that are parallelizable. Intel TBB’s many performance features make it a good choice for improving this code. In addition to providing parallelism, the program can take advantage of Intel TBB concurrent containers to store data (in Lab 2) and use the Intel TBB scalable memory allocator to improve the significant memory handling.

**Instructions:**  
Before attempting this lab please make sure you have watched the Introduction to Threading Building Blocks video found at http://software.intel.com/en-us/videos/introduction-to-intel-threading-building-blocks/ to learn about Intel TBB.

**Part 1: Create a baseline.**  
1. Copy the file TBBLab.zip from PBB Labs/Threading Building Blocks to your local directory.  
2. Unpack TBBLab.zip into your local directory by right-clicking the file and choosing Winzip > Unzip.  
3. Open the TBBLab.sln solution in Microsoft Visual Studio 2008*. Note that this is pre-configured, so you shouldn’t need to change anything (Figure 4).  
4. Verify that you are using the win32 Release version of the solution (Figure 5).  
5. Rebuild the solution using Build > Rebuild and verify that it builds successfully (Figure 6).
Figure 6

6. The code for Lab 1 uses a database of records stored as std::vector, each containing a first name, last name, and age. The vector is sorted by age, and then the number of occurrences of each name is stored in a std::map container called “stats.” Although it is not the focus of this lab, this program uses an Intel TBB parallel_for template to count the names in parallel. You can (optionally) examine the code for this starting in the CountNames function in main.cpp.

7. Select Debug > Start without Debugging to run the program.

8. Record the total run time and the sort run time. Make sure you are running the Release build for performance measurements.

Sort run time: __________________________________________

Total run time: __________________________________________

9. Now optimize the program so that both the sort and total time are improved.

Part 2: Optimize the sort.

1. The sorting of the names is done by calling the Standard Template Library (STL) sort function on line 44 in main(). To improve the sort time (in the sort function called from main()), use the Intel TBB parallel_sort template documented in section 4.11 of the reference guide. Note that Intel TBB parallel_sort takes the same parameters as STL sort. Don’t forget to include tbb/parallel_sort.h.

2. Build the program using Build > Build Solution, and then re-run the program using Start > Start without Debugging. Record the new total run time and sort run time.

Sort run time: __________________________________________

Total run time: __________________________________________
3. What was your speedup from using a parallel sort instead of a serial sort?

**Part 3: Optimize the memory management.**

1. The program's run time can be further improved by using the Intel TBB scalable memory allocator documented in section 10 of the tutorial. To do this, use `tbb_allocator<char>` as the last parameter to type `MyString` in `interface.h` line 21. (Currently, the last parameter is blank, which means that the STL standard allocator is being used.) By doing this, you instruct STL strings to allocate and deallocate memory using the Intel TBB scalable allocator. Don’t forget to include `tbb/tbb_allocator.h`.

2. Re-run the program and record the new total run time and sort run time.

   **Sort run time:**
   ____________________________________________________________

   **Total run time:**
   ____________________________________________________________

3. What was your speedup after using the Intel TBB scalable allocator?

**Conclusion:** Using Intel TBB to improve this code resulted in significant scalability, without requiring you to manage threading details. Intel TBB will automatically attempt to utilize all the processing cores on the system where it is being run (by default).

**Exercise 2: Sorting and Counting using Intel® Threading Building Blocks**

**Difficulty Level:** Intermediate  
**Lab Project Setting:** Microsoft Visual Studio 2008* (64-bit project)  
**Materials Needed:**

**Objective:** This lab uses the same program as Intel TBB Lab 1 (a simple application that counts and sorts a database of personal information, stored as std::vector). In this exercise, you will use Intel TBB to further optimize the counting of names.

Why use this PBB? This program uses a central data structure, allocates and deallocates significant amounts of memory, and manipulates vectors in ways that are parallelizable. Intel TBB's many performance features make it a good choice for improving this code. In addition to providing parallelism, the program can take advantage of Intel TBB by using its concurrent containers to store data, its atomics to ensure thread safety with low overhead, and its scalable memory allocator to improve the significant memory handling.

**Instructions:**

**Part 1: Create a baseline.**

1. For this lab, you will continue using the Intel TBB lab project optimized in Lab 1. The code uses an Intel TBB `parallel_for` template to count the names. Threads work in parallel to go through the database of names. It looks up each one in an STL map container.

   The map contains the names, each stored with a value that is the number of occurrences of the name. Each time a name is found in the database, the number of occurrences is updated in the map. See the code in the CountNames function in `main.cpp` to understand how this works.

2. Select **Debug > Start without Debugging** to run the program.

3. Record the total run time and the sort run time.

   **Sort run time:**
   ____________________________________________________________

   **Total run time:**
   ____________________________________________________________

4. Now optimize the program so that the time for the count (which contributes to the total run time) is improved.

**Part 2: Optimize the count.**

1. Look again at the code in the CountNames function in `main.cpp`. A Windows*-critical section object is used to ensure that two threads cannot both access the map at the same time. (For more on critical sections, see Microsoft Visual Studio 2010* Help.) When using templates like Intel TBB `parallel_for`, the user is responsible for protecting data that might be accessed by multiple threads. However, note that using global locks in parallel programs usually limits performance and scalability. Instead of using a data container that requires a global lock for thread-safe access,
switch to concurrency-friendly containers, such as Intel TBB concurrent containers, that minimize or avoid synchronization.

2. Use an Intel TBB concurrent_unordered_map container instead of the STL map for the Stats type declared in interface.h line 23. The concurrent_unordered_map container supports concurrent insertion and traversal, meaning that multiple threads can add new entries and look up values without data corruption. See Section 5.2 of the Intel TBB reference guide. Don’t forget to include tbb/concurrent_unordered_map.h.

3. Using a concurrent container means that accessing the map no longer needs to be protected by a global lock. But updates to the counter values stored in the stats map still need protection – otherwise, a thread could be reading a count value at the same time as another thread was writing it. Since the update is a simple integer addition, the easiest and most lightweight way to ensure safety is by using atomic operations. Use an Intel TBB atomic as the type for the values in the Intel TBB concurrent_unordered_map by replacing the second parameter of the map (currently int) with atomic<int>. (See type Stats on line 23 of interface.h.) Section 8 of the Intel TBB Tutorial discusses atomics. Don’t forget to include tbb/atomic.h.

4. Now that a concurrent container and atomic updates are being used, the critical section is no longer needed. Don’t forget to remove the critical section calls in the CountNames function in main() (in main.cpp).

5. Build the program using **Build > Build Solution.** Select **Debug > Start without Debugging** to run the program.

6. Ensure the program is correct—it should print out “Correct!” at the end if the sort and count were validated.

7. Record the total run time and the sort run time.

Sort run time:
_____________________________________

Total run time:
_____________________________________

8. What was your speedup after removing the overhead of the critical section? Compare to part 1, step 3.

**Conclusion:** Using Intel TBB to improve this code resulted in significant scalability, without the developer needing to manage threading details. Intel TBB’s building blocks provide developers with a choice of pre-coded components to meet their parallelism needs.
Intel® Cilk™ Plus

What is it? What are the benefits?

Intel® Cilk™ Plus has five components: Cilk Plus keywords and reducers, array notations, elemental functions, and user-mandated vectorization. They are integrated into Intel Parallel Composer 2011. Intel Cilk Plus is the easiest way to parallelize a serial program. Its simple syntax is easy to learn and use.

- Leverage unambiguous semantics with a strict fork-join model.

Intel Cilk Plus provides the easiest way for you to understand the parallel control flow of your program.

- Benefit from automatic load balancing via work stealing, and serial equivalence due to parent stealing.

- The low overhead task spawning encourages creation of many small tasks. A program with many small tasks provides an opportunity for the task scheduler to both load balance and forward scale to larger core counts.

When should you use Intel® Cilk™ Plus keywords and reducers?

Use cilk_spawn/cilk_sync/cilk_for/reducers when:

- You need the easiest and fastest way to parallelize a serial program without any dependencies using Intel Composer 2011. The simple syntax is easy to learn and use.

- You want to quickly guarantee serial equivalence (i.e., get the right answer every time) due to parent stealing.

- Data parallel constructs are not particularly useful for the problem at hand and you are coding up something that doesn’t map to an already-existing algorithm implementation provided in Intel TBB.

---

<table>
<thead>
<tr>
<th>Key Features</th>
<th>Feature</th>
<th>Example</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spawning a function call</td>
<td>x = cilk_spawn func(g(y),h(z));</td>
<td>func executes asynchronously.</td>
</tr>
<tr>
<td></td>
<td>Synchronization statement</td>
<td>cilk_sync;</td>
<td>Wait for all children spawned inside the current function.</td>
</tr>
</tbody>
</table>
|             | Parallel_for loop | cilk_for (int i = 0; i < N; i++) {
|             |                      | statement;
|             |                      | } | Loop iterations execute in parallel. |
Array Notations

What is it?
Array Notations are a language extension that allows existing language operators on arrays of existing data types.
- Its syntax is very amenable to specifying array sections.
- Define element-wise operations on array elements.
- You can map scalar functions onto multiple array elements at once via elemental functions.
- Array Notations also offers robust reduction operators.

Key features
Array sections
A language extension that allows existing language operators on arrays of existing data types:
<array base> [:<lower bound>:<length> [:<stride>]]
Section specifier is [lower-bound:length] pair (memcpy style) as opposed to Fortran-style [lower:upper]
- A[] // All of vector A
- C[][5] // Column 5 of matrix C
- D[0:3:2] // Elements 0,2,4 of vector D

Operator maps
Most C/C++ arithmetic and logic operators are available for array sections.
+,-,*,/,%,<,==,>,<,>=,>,++,||,&&,l,
{unary},+(unary),++,-=,*,/=,(pointer de-referencing)
Operators are implicitly mapped to all elements of the array section operands.
a[] * b[] // element-wise multiplication
Operations on different elements can be executed in parallel without any ordering constraints.
Array operands must have the same rank and size.
a[0:4][1:2] + b[1:2][0:4] // error, different rank sizes
The scalar operand automatically expanded to fill the whole section.
a[0:4][1:2] + b[0][1][1] // ok, adds a scalar b[0][1]

Assignment maps
The assignment operator applies in parallel to every element of the array section on LHS. The LHS of the assignment defines the array context where RHS is evaluated. The rank of the RHS array section must be the same as the LHS. The length of each rank must match the corresponding LHS rank. Scalar is expanded automatically.
a[][] = b[][2][] + c;
e[] = d;
e[] = b[][][1][1] // error, different rank
a[][] = e[] // error, different rank
RHS is evaluated before any element on LHS is stored. The compiler can vectorize the RHS computation even if some operand on the RHS may alias with the L-value on the LHS.
a[1:s] = a[0:s] + 1; // use old value of a[1:s-1]

Reductions
Reduction combines array section elements to generate a scalar result.
Int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[]); // sum is 10
There are nine built-in reduction functions supporting basic C data-types.
__sec_reduce_add __sec_reduce_mul
__sec_reduce_all_zero __sec_reduce_all_nonzero
Parallelism across processor achieve two conquer

Why use this PBB

100x100 matrix transpose.

Intel Cilk Plus to improve the processor cores. At keyword

Difficulty Level

Summation Using Intel Cilk Plus

Exercise 3: Matrix Transpose and Square Root Summation Using Intel Cilk Plus

Instructions:

Before attempting this lab please make sure you have watched the Cilk Plus 5min Intro video found at http://software.intel.com/en-us/videos/introduction-to-intel-cilk-plus/ to learn about Intel Cilk Plus.

Part 1: Create a baseline.

You have C++ code that does two matrix transposes, one on an array of doubles and one on the result of the first transpose as part of a correctness check. This code is defined in matrix-transpose.cpp. Nested in the transpose code when each swap of two doubles occurs is a function that does a summation of square roots. It then peels back the summation to again do a correctness check of the summation. This code is in sum_squareroot.cpp, sum_squareroot.h, subtract_squareroot.cpp, and subtract_squareroot.h.

1. Copy matrix-transpose.zip from C:/IDF LAB/Cilk Plus to your local directory.
2. Unpack matrix-transpose.zip into your local directory by right clicking the file and choosing Winzip > Extract.
3. Open the matrix-transpose.sln solution in Microsoft Visual Studio 2008*. You can do this via the File menu under Open > Project/ Solution. Note that this is pre-configured, so you shouldn’t need to change anything (Figures 7 and 8).

Confirm that the project is in Release x64 mode and that you can see the three header files: cilktime.h, subtract_squareroot.h, and sum_squareroot.h. Also, make sure that you can see the three source files, which include matrix-transpose.cpp, sum_squareroot.cpp, and subtract_squareroot.cpp (Figure 9).

4. Rebuild the solution using Build > Rebuild Solution. Confirm there are no errors (Figure 10).

5. Run the resulting matrix-transpose.exe. You can do this by pushing Ctrl+F5 from the Microsoft Visual Studio* IDE or by selecting Debug > Start without Debugging. Confirm there are no runtime errors, and record the time taken to execute.

First transpose time: ________________

Second transpose time: ________________
Part 2: Optimize the main transpose loop.

You have been assigned the project to optimize this code to take advantage of parallel opportunities across multiple cores. Previous performance analysis has shown that the key hotspot of this code is in the `sum_squareroot()` function in `sum_squareroot.cpp`. To improve performance, you need to distribute the work from the multiple calls of `sum_squareroot()` from the `matrix_transpose()` function defined in `matrix-transpose.cpp` across multiple CPU cores. Do the following to take advantage of task parallelism to increase performance.

1. Open the Intel® Composer User’s Guide via the link in the Materials section of this lab. Refer to the description of the keywords in the user’s guide under Creating Parallel Applications > Using Intel® Cilk™ Plus > Intel® Cilk™ Plus Language Features > Keywords. Also, read the description of `cilk_for` in the user’s guide under Creating Parallel Applications > Using Intel® Cilk™ Plus > Intel® Cilk™ Plus Language Features > `cilk_for`.

2. The body of `matrix_transpose()` consists of a nested for loop that iteratively calls `sum_squareroot()` and then swaps two items in the array of doubles to perform the matrix transpose. The outer for loop can be run in parallel because it is data-independent, meaning that the results of one iteration do not depend on a previous iteration, and there are no potential data races. Add a `#include <cilk/cilk.h>` and replace the outer for loop in `matrix_transpose()` with a `cilk_for` to run the individual iterations of the for loop in parallel across multiple CPU cores.

3. Build the application using Build > Build Solution, and then run the application again using Debug > Start without Debugging or Ctrl+F5. Verify that multiple cores are being used by running Task Manager (right-click on the Start bar at the bottom of the screen and select “Start Task Manager”) and looking at the performance tab. Verify the resulting executable’s correctness and record the performance increase.

First transpose time: __________________________________________________________
Second transpose time: _______________________________________________________

4. What was your speedup after parallelizing the main transpose loop? (Compare to Part 1, step 6.)

**Conclusion:** Intel Cilk Plus gives substantial performance and scaling benefits with little time and effort. With one line of code changed, we can observe almost linear scaling.

**Exercise 4: Matrix Transpose and Square Root Summation Using Intel Cilk Plus**

**Difficulty Level:** Advanced  
**Lab Project Setting:** Microsoft Visual Studio 2008* (64-bit project)  
**Materials Needed:**  
- Intel C++ Compiler User’s Guide  
- Available from C:\Program Files (x86)\Intel\Parallel Studio 2011\Composer\Documentation\en_US\compiler_c/c\index.htm (Assumes you have installed Intel Parallel Studio 2011)

**Objective:** This lab introduces you to using the Intel Cilk Plus array notations and the elemental function `__declspec(vector)` to enable the Intel Compiler to create the most efficient code for your program. At the end of this lab, you will have used Intel Cilk Plus to improve the performance of a loop doing a 100x100 matrix transpose.

**Why use this PBB?** When you have for loops or divide-and-conquer algorithms, Intel Cilk Plus provides a simple API to achieve two different types of performance gains. Parallelism across processor cores can be achieved by using the Intel Cilk Plus parallelism keywords such as `cilk_for` (Lab 1). Within a single core, data parallelism can be achieved by using the Intel Cilk Plus array notations or elemental functions (Lab 2), which enable the usage of vectorization hardware on the processor.

**Instructions:**

**Part 1: Optimize the Summation.**

You have been assigned the project to optimize this code to take advantage of parallel opportunities across multiple cores. Previous performance analysis has shown that the key
hotspot of this code is in the sum_squareroot() function in
sum_squareroot.cpp. You need to use Intel Cilk Plus array
notations and elemental functions to optimize these
operations with SIMD processor instructions.

1. For this lab you will continue using the matrix-transpose
project optimized in Lab 1. Start by re-running matrix-
transpose.exe (using Debug > Start without Debugging) and
recording the time taken to execute.

First transpose time:
____________________________________

Second transpose time
____________________________________

2. Open the Intel® Composer User’s Guide via the link in the
Materials section of this lab. Refer to the description of Intel
Cilk Plus array notation in the user’s guide under Intel C++
Compiler 12.0 User and Reference Guide > Creating Parallel
Applications > Using Intel® Cilk™ Plus > Extensions for Array
Notation > C/C++ Extensions for Array Notations
Programming Model. Focus especially on the syntax of the
initial examples. For more background, you may want to also
read the overview at Creating Parallel Applications > Using
Intel® Cilk™ Plus > Extensions for Array Notation > C/C++
Extensions for Array Notations Overview.

3. For the first for loop in sum_squareroot(), the body is a
sequence of array accesses along with the sqrt() math
function call. You can use Intel Cilk Plus array notation to
allow the compiler to generate more efficient data-parallel
code for this for loop. Replace the inner for loop (with the
loop index j) with an array-wide += increment and
assignment with lower bound 0 and length size and a stride
of 1.

4. Build the application using Build > Build Solution, run it
using Debug > Start without Debugging, verify the resulting
executable’s correctness, and record the performance
increase.

First transpose time:
____________________________________

Second transpose time:
____________________________________

5. What was your speedup after using array notations to
enable compiler vectorization? (Compare to Part 1, step 1.)

Part 2: Optimize the subtraction.

1. Refer to the description of Intel Cilk Plus elemental
functions and __declspec(vector) in the user’s guide at
Creating Parallel Applications > Using Intel® Cilk™ Plus >
Elemental Functions.

2. In the second for loop in sum_squareroot.cpp, the for loop
calls an external function subtract_squareroot(), which is a
simple scalar function. You can define this function as an
Intel Cilk Plus elemental function so that the compiler can
apply this function to multiple elements of the array at one
time, vectorizing the code. This is done using the
__declspec(vector) notation on both the declaration and
definition of subtract_squareroot. Make the code changes
necessary (in subtract_squareroot.cpp and .h).

3. Build the application using Build > Build Solution, run it
using Debug > Start without Debugging, verify the resulting
executable’s correctness, and record the performance
increase.

First transpose time:
____________________________________

Second transpose time:
____________________________________

4. What was your speedup after using an elemental function
to allow the compiler to apply subtract_squareroot to
multiple array elements in parallel?
(Compare to Part 1, step 4.)

Conclusion: Intel® Cilk™ Plus gives substantial improvement
and scaling benefits with little time and effort. With one line
of code removed, and three lines changed, we can observe a
significant performance benefit.
Intel Array Building Blocks (Intel ArBB)

Intel ArBB can be defined in a number of ways. First and foremost, it is an API, backed by a library. No special preprocessor is needed to work with ArBB. It is a programming language extension (i.e., an "attached language" that requires a host language).

Intel ArBB extends C++ for complex data parallelism, including irregular and sparse matrices. It has the following characteristics:

- A portable parallel development platform
- Hardware-independent parallel computations
- Sequential semantics, good data locality
- Safety by default: no deadlocks, no race conditions

Intel ArBB is best suited for compute-intensive, data-parallel applications (often involving vector math).

Next, this API gives rise to a generalized data-parallel programming solution. It frees application developers from dependencies on particular hardware architectures. It integrates with existing C++ development tools and allows parallel algorithms to be specified at a high level.

Intel ArBB is based on dynamic compilation that can remove modularity overhead. It translates high-level specifications of computations into efficient parallel implementations, taking advantage of both SIMD and thread-level parallelism.

Key features

Any Intel ArBB program experiences two compilations (Figure 11). One is C++ compilation for Intel® Architecture (IA) binary distribution. The other is dynamic compilation for high-performance execution. The second compilation is done by the Intel ArBB Dynamic Engine. You need to copy data from the C++ space to the Intel ArBB space in order for that data to be optimized for multiple cores. Data is kept in an isolated data space and is managed by the collection classes.

Take a look at the regular and irregular data structures that are intrinsic to the platform (Figure 12). You express your computation over those structures as C++ functions using operations over Intel ArBB collections and/or scalar types.
Exercise 5: Modular Computation on Large Arrays Using Intel® Array Building Blocks

Difficulty Level: Beginner
Lab Project Setting: Microsoft Visual Studio 2010* (64-bit project)

Objective: This lab introduces you to using Intel® Array Building Blocks' (Intel® ArBB) dense containers and functions. The program uses dense containers to store two arrays of 1024 elements. In this exercise, you will write a function to compute the sum of the squared difference of these two arrays and use Intel ArBB to invoke that function in parallel in a way that takes advantage of the vectorization and SIMD capabilities of the processor being used.

Why use this PBB? Written using traditional programming methods, this program would create two large arrays and use loops to serially perform computations on them. But since the computations are parallelizable and vectorizable, Intel ArBB can be used to create the program from scratch with far less code and much better performance. The high level of abstraction provided by Intel ArBB allows a developer to think in terms of the whole arrays rather than having to manipulate individual elements. Using Intel ArBB's runtime means that the application will automatically get the benefit of whatever parallelization and vectorization can be applied using the hardware available.

Instructions:

Before attempting this lab please make sure you have watched the ArBB_Smin_Intro video found at http://software.intel.com/en-us/videos/introduction-to-intel-array-building-blocks/, to learn about Intel ArBB.

Part 1: Set up the solution.
1. Copy the file ArBBLab.zip from C/IDF LAB/Array Building Blocks to your local directory.
2. Unpack ArBBLab.zip into your local directory by rightclicking the file and choosing WinZip > Unzip.
3. Inside the ArBBLab folder, run the tutorial.bat file to open the solution in Microsoft Visual Studio 2010*.
4. A Microsoft Visual Studio solution should appear. This solution has already been configured to use ArBB, but it will not compile yet.
5. Open the file tutorial.cpp in the source code editor. Note that the correct headers and name-space declaration to use Intel ArBB have already been included.
6. Read through the code and comments for the main function in tutorial.cpp. Intel ArBB is a new programming model that requires you to think differently (but in a more intuitive way) about arrays and array operations. In order to use Intel ArBB, you need to do three things:

   ➢ Create data structures that will be stored within Intel ArBB's memory space. This has been done for you. The main code creates two Intel ArBB dense containers, a and b, and fills them with sample data.
   ➢ Record the operations you want done on those containers. Intel ArBB’s run-time component will compile those operations into efficiently parallelized, vectorized code and will then cache the code so that it can be run many times. You will do this in Part 2.
   ➢ Properly invoke the operations on the Intel ArBB containers. You will do this in Part 3.

Part 2: Record the operations to be done on your dense containers.

1. You must define the computation to be executed on the dense containers a. and b. You do this using a normal C++ function that conforms to Intel ArBB guidelines. The function for this lab will be called sum_of_squared_differences and is declared on line 24. Go to this function and notice that, as required by Intel ArBB, its return type is void, and the variable to hold the result (called result) is passed by reference.

2. Now you can record the operations to be done on the data. Intel ArBB allows you to think in terms of the whole container rather than the individual elements. This enables you to write this computation in terms of the containers, a and b, and store the result in another container that you create. The operation to be done on these containers is to compute the squared difference, just like in the Intel ArBB introduction video. Go to line 32 of tutorial.cpp and write the line of code to compute the squared difference of the Intel ArBB dense containers and store it in a third dense container called “temp.” There are comments on lines 26-31 to help you, or you can review the video at around time 3:59.

3. Now you will add one more operation. The example in the Intel ArBB video computed the squared difference between two Intel ArBB dense containers and stored it into a third Intel ArBB dense container. Remember that in standard programming, dense containers would be arrays. At the end of step 2 above, and in the example in the video, there is a dense container holding the squared difference of each element in the array a and its corresponding element in b. The program in this lab will compute the SUM of the squared differences of a and b. To do this, it needs to go through every value in the dense container storing the squared differences (called “temp” in tutorial.cpp), and sum up those values.

Summing an array is a type of reduction, meaning that an operation is repeatedly applied to a set of values to “reduce” them down to one value. This can be done in parallel using the Intel ArBB function called add_reduce(). Instead of
Exercise 6: Intel® Array Building Blocks Scoped Timer and Run-time Compiler in Action

**Difficulty Level:** Intermediate

**Objective:** This lab gives you more hands-on experience with Intel ArBB's dense containers and functions. You will also learn how to use a scoped timer to experiment with the run-time compiler and see it in action.

**Why use this PBB?** This program uses an Intel ArBB function to capture a basic arbitrary computation across two large arrays, in this case a subtraction/multiplication and an add-reduction. One does not have to address each of the individual elements in the array during the computation through any sort of loop. The operations are global across all of the data elements in parallel and the run-time compiler will utilize all available resources to execute the computation in parallel. In this lab, we will see how to time the execution time for each run of the Intel ArBB function and see exactly when the runtime compilation takes place. Intel ArBB functions allow you to create custom algorithms where you specify what you would like done to your arrays, not how to do it. Intel ArBB generates vectorized and threaded code tailored for the architecture in question.

**Instructions:**

**Part 1:** Change the size and contents of your dense containers.

1. For this lab you will continue using the same array computation program written in Lab 1. In Part 1, you will modify the code to get more experience with Intel ArBB dense containers. Begin by modifying the size of the dense containers. Choose any size you want, not exceeding 1024 * 200 (for memory reasons), and change the value of the size on line 49.

2. Next, change your input elements to the dense containers. While Intel ArBB dense containers allow you to interact with a whole data structure at once, you can also iterate through them like traditional arrays using a read_write_range. See lines 59-60 where a read_write_range has been created for each dense container.

3. In Lab 1, the dense containers were filled with 1s (for a) and 2s (for b) on lines 63 and 64. For this lab, you can practice filling the containers with input as if they were traditional arrays. You do this by iterating through the read_write_range. Start by removing lines 63 and 64. (What should be on these lines is two std::fill calls.)
4. Now, around line 63, create your own code to iterate through the range of each dense container and assign initial values to the data. You can fill them with any input you want. Remember, however, to make the data 32-bit floats since the dense containers are of Intel ArBB type f32. Below is an example of how to iterate through the range assigning values. You can use the same values assigned below, or your own.

```cpp
const double multiplier = 2.0;
for (std::size_t i = 0; i != size; ++i) {
    range_a[i] = static_cast<float>(multiplier * i);
    range_b[i] = static_cast<float>(multiplier * i * 2.0);
}
```

**Part 2:** Conduct timing experiments.

1. For this part of the lab, you will be replacing part of the code in main. To begin, you need to remove the code from Lab 1 that executed the Intel ArBB function and outputted the result to the screen. This should be all code in main after the creation of the f64 result variable and before the catch clause. Look for the comments marked with "****" in main and delete all code between them.

2. Read through the code below that will run the Intel ArBB function multiple times and time its execution. Then copy the code below into main in the area where you just deleted Lab 1 code. Make sure to fill in the line where indicated to invoke the sum_of_squared_differences function.

```cpp
double time = std::numeric_limits<double>::max();
//modify number of iterations below if desired
//(initially set to 10 in the for loop)
for (int i = 0; i < 10; i++) //for loop
double time_i;
    { // profiling section using scoped_timer
    const scoped_timer timer(time_i);
    //<fill in - call sum_of_squared_differences ArBB function,
    //just like in Lab 1, part 3>
    }
    //scoped timer
    time = std::min(time, time_i);
    std::cout << "Time: " << (i + 1) << " ms\n";
} //for loop
std::cout << "n";
std::cout << "Time: " << time << " ms\n";
std::cout << "Result: " << value(result) << \n";
} //try
```

3. After filling in the call to the Intel ArBB function in step 3, build the code using **Build > Build Solution** and fix any errors.

4. Run the code using **Debug > Start without Debugging**. What were your results?

   **Number of calls to Intel ArBB function:**
   _______________________________________________________________

   **Minimum time to execute:**
   _______________________________________________________________

   **Execution time of first call:**
   _______________________________________________________________

5. The Intel ArBB run-time compiler creates an efficient compiled version of your Intel ArBB function and then caches that compilation for future use. By calling the function repeatedly, you are seeing the effects of the run-time compiler’s behavior. To finish up this lab, try experimenting with the number of runs, data size, or operations done on the arrays inside the Intel ArBB function. You are also welcome to switch between the Intel and Microsoft Visual Studio compilers. Try to figure out why you see any differences in timings.

**Conclusion:** Intel ArBB provides developers with a variety of ways to structure their data and record computations on that data. Intel ArBB’s run-time library will compile Intel ArBB functions into their most parallel and efficient versions and then cache that compilation to maximize performance.

**What to take home regarding Intel PBB**

Intel PBB represents the next generation of parallel programming models.

Intel TBB is popular, proven, widely used, and recommended by top companies in the industry. Intel ArBB is an emerging technology currently in beta and is built using the Intel TBB runtime for threading and its own runtime for generating SIMD instruction sets tailored for the architecture. Intel Cilk Plus is innovative technology built into the Intel Compiler to simply and robustly take advantage of parallel opportunities. Use Intel PBB for all new development where other models are not in use.

See below for more information regarding performance and optimization choices in Intel software products.
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101