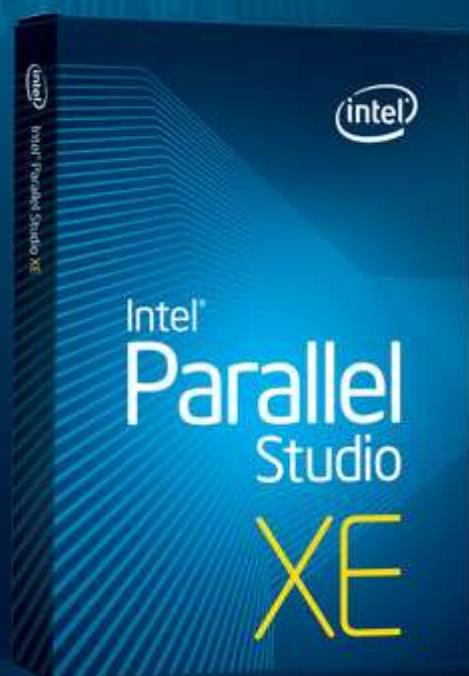




A Simple Path to Parallelism with Intel® Cilk™ Plus



This document is an introductory tutorial describing how to use Intel® Cilk Plus to simplify making your code take advantage of vectorization and threading parallelism. It provides a brief description of the goals of the product feature and walks through an end-to-end example showing how it is used.

Intel Cilk Plus is part of the Intel® C++ compiler that's available in Intel® Studio XE product suites.

A Simple Path to Parallelism with Intel® Cilk™ Plus

Compiler extensions to simplify task and data parallelism

Intel® Cilk™ Plus adds simple language extensions to express data and task parallelism to the C and C++ language implemented by the Intel® C++ Compiler, which is part of Intel® Studio XE product suites and Intel® Composer XE product bundles. These language extensions are powerful, yet easy to apply and use in a wide range of applications. Intel Cilk Plus has several benefits including:

Feature	Benefit
Simple keywords	Simple, powerful expression of task parallelism: <ul style="list-style-type: none"> • <code>cilk_for</code> - Parallelize for loops • <code>cilk_spawn</code> - Specify the start of parallel execution • <code>cilk_sync</code> - Specify the end of parallel execution
Hyper-objects (Reducers)	Eliminates contention for shared reduction variables amongst tasks by automatically creating views of them for each task and reducing them back to a shared value after task completion
Array Notation	Data parallelism for whole arrays or sections of arrays and operations thereon
Elemental Functions	Enables data parallelism of whole functions or operations which can then be applied to whole or parts of arrays

Intel Cilk Plus has an [open specification](#) so other compilers may also implement these exciting new C/C++ language features.

How is the Performance and Scaling?

Here are two real application examples. One is a Monte Carlo simulation utilizing Intel Cilk Plus, where the array notation allows the compiler to vectorize, that is, utilize Intel® Streaming SIMD Extensions (Intel® SSE) to maximize data-parallel performance, while adding the `cilk_for` causes the driver function of the simulation to be parallelized, maximizing use of the multiple processor cores for task-level parallelism. The second is a Visual Computing algorithm using the array notation to vectorize the render function and using `cilk_for` for task parallelism to distribute the work across multiple cores. Both applications get a dramatic speedup with very little effort! Compiler: Intel Parallel Composer XE 2013 with Microsoft® Visual Studio® 2010. ¹System Specifications: Intel® Core™ i5-3550 processor, 3.3 GHz, 4 cores, 4GB RAM, Microsoft® Windows® Server 2008 R2 Enterprise x64, service pack 1.

	Scalar Code	Intel Cilk Plus Data-Parallel	Intel Cilk Plus Task-Parallel	Intel Cilk Plus Data and Task-Parallel
Monte Carlo Simulation ¹	9.31 sec	4.62 seconds = 2.02x speedup	2.51 sec = 3.71x speedup. (4 cores; no hyperthreading)	1.25 seconds = 7.45x speedup
AOBench	1.96 frames/second	4.52 frames/second = 2.31 x speedup	6.73 frames/second = 3.44x speedup	14.68 frames/second = 7.50x speedup

When to use Intel Cilk Plus over other Parallel Methods?

Use Intel Cilk Plus when you want the following:

- Simple expression of opportunities for parallelism, rather than control of execution to perform operations on arrays
- Higher performance obtainable with inherent data parallelism semantics - array notation
- To use native programming, as opposed to managed deployment: no managed runtime libraries - you express the intent to mix parallel and serial operations on the same data

Intel Cilk Plus involves the compiler in optimizing and managing parallelism. The benefits include:

- Code is easier to write and comprehend because it is better integrated into the language through the use of keywords and intuitive syntax
- The compiler implements the language semantics, checks for consistent use and reports programming errors
- Integration with the compiler infrastructure allows many existing compiler optimizations to apply to the parallel code. The compiler understands these four parts of Intel Cilk Plus, and is therefore able to help with compile time diagnostics, optimizations and runtime error checking.



A Simple Path to Parallelism with Intel® Cilk™ Plus

Try It Yourself

This guide will help you begin adding Intel Cilk Plus to your application using Intel Parallel Studio XE 2013. It will show you the following examples:

- a simple quick-sort implementation with the `cilk_spawn` and `cilk_sync` keywords
- Monte Carlo Simulation, and a Visual Computing algorithm implementation to show both the array notation syntax and the `cilk_for` keywords
- A reference to a simple Getting Started Tutorial showing how to use Intel Cilk Plus SIMD Vectorization and Elemental Functions

Install Intel Parallel Studio XE

Install and Set Up Intel® Parallel Studio XE

Estimated completion time: 15-30 minutes

1. [Download](#) an evaluation copy of Intel Parallel Studio XE.
2. Install Intel Parallel Studio XE by clicking on the `parallel_studio_xe_2013_setup.exe` (can take 15 to 30 minutes depending on your system).

Note: The examples in this guide use Microsoft* Visual Studio* on Windows*, but Intel Cilk Plus is also available for Linux* in the Intel® C++ Compiler in Intel Parallel Studio XE for Linux*.

Get the Sample Applications

Install the sample applications:

1. Find the Intel Cilk Plus Samples, `Cilk.zip`, in the installation directory of Intel Parallel Composer. In a typical installation, this is `c:\Program Files (x86)\Intel\Composer XE 2013\Samples\en_US\C++\Cilk.zip`.
2. Download the “[MonteCarloSample](#)” sample file to your local machine. This sample shows a serial/scalar kernel for the Monte Carlo simulation, an Intel Cilk Plus `cilk_for` version of the driver function, an Intel Cilk Plus array notation version of the kernel, and another version using both `cilk_for` and the array notations.
3. Download the “[CilkPlus-AOBench](#)” sample file to your local machine. This sample uses an algorithm called “ambient occlusion” to draw lighting and shadows on three moving spheres. The sample shows a serial/scalar implementation of the algorithm, an Intel Cilk Plus Array Notation version to vectorize the render code for data parallelism, an Intel Cilk Plus `cilk_for` version for task parallelism to split the work across multiple cores, but does not use data parallelism, and a version that takes advantage of both task and data parallelism, multiplying the speedup from each. The program draws an animation with complex lighting effects, rendered in real-time. The animation is rendered four times, each with a different code implementation. This program demonstrates the large performance gains possible with Intel® Cilk™ Plus.
4. Extract the files from each zip file to a writable directory or share on your system, such as a `My Documents\Visual Studio 200x\Intel\samples` folder.
5. After extracting all of the samples, you should see the following directories:
 - `Cilk` - with multiple sample directories underneath it. We will use `qsort`.
 - `MonteCarloSample`
 - `CilkPlus-AOBench`

A Simple Path to Parallelism with Intel® Cilk™ Plus

Build the Samples:

Each sample has a Microsoft* Visual Studio* 2005 solution file (.sln) that can be used with Visual Studio 2005, 2008 and 2010.

1. Build the solutions with Intel Parallel Composer XE 2013, using the Intel C++ Compiler in the Release (optimized) configuration settings.
2. In each of these solution files, the Intel Cilk Plus language extensions are enabled.
 - a. Right-click the project, select Properties and expand the Configuration Properties > C/C++ > Language .
 - b. Set the "Replace Intel Cilk Plus Keywords with Serial Equivalents" to No.
 - c. Set the "Disable All Intel Language Extensions" to No. Following are the Configuration Properties for qsort as shown in figure 1.

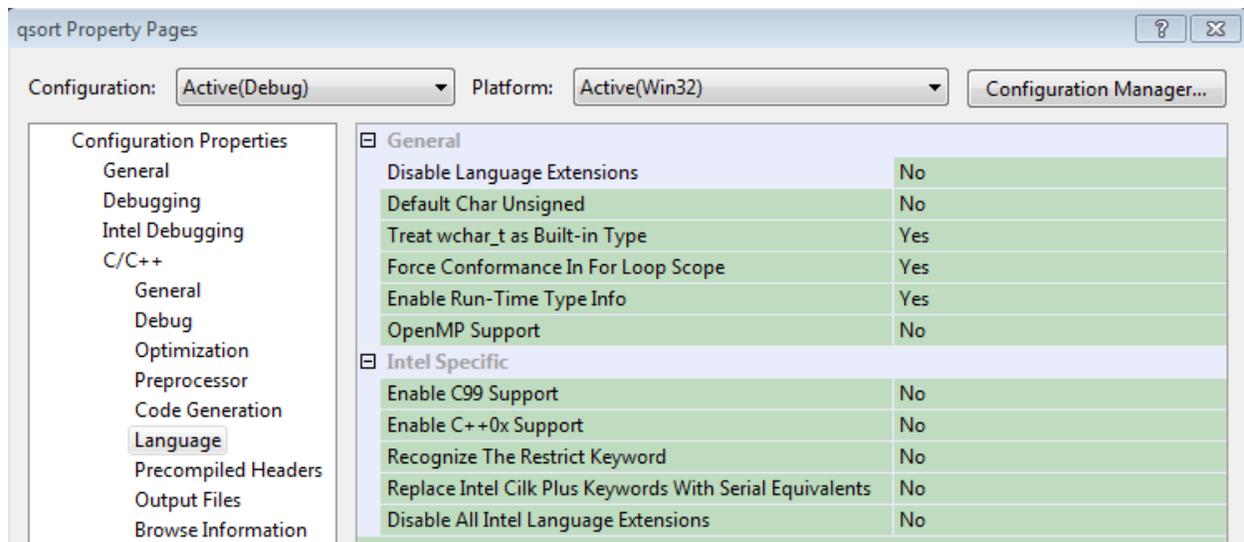


Figure 1

3. Run the applications from within Microsoft Visual Studio. Go to **Debug > Start Without Debugging**

Implement Parallelism with the Intel Cilk Plus Keywords

Now, we are going to quickly add task parallelism using the keywords *cilk_spawn* and *cilk_sync*.

1. Load the qsort solution into Microsoft Visual Studio.
2. Open the **qsort.cpp** file and look at **sample_qsort()** routine shown in figure 2.

A Simple Path to Parallelism with Intel® Cilk™ Plus

```

void sample_qsort(int * begin, int * end)
{
    if (begin != end) {
        --end; // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                                     std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle); // move pivot to middle
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end); // Exclude pivot and restore end
        cilk_sync;
    }
}

```

Figure 2

Look at the use of the *cilk_spawn* and *cilk_sync*. To parallelize the quick-sort algorithm, the *cilk_spawn* will take care of the task creation and scheduling of tasks to threads for you, while the *cilk_sync* indicates the end of the parallel region at which point the tasks complete and serial execution resumes. In this example, the two calls to *sample_qsort()* between the *cilk_spawn* and *cilk_sync* can be executed in parallel, depending on the resources available to the Intel Cilk Plus runtime.

At the time of this writing, Intel Cilk Plus is also available in an [open source branch to gcc](#). We have published the specification to encourage other tools vendors to adopt it (see Additional Resources below). A key property of the Intel Cilk Plus keywords is that even if you disable them, their serial semantics guarantees the application will still run correctly in serial mode—without modifying the code! Thus, you can easily check serial and parallel runtime performance and stability by disabling/enabling them in the Property Pages as seen in the previous section: Replace Intel Cilk Plus Keywords With Serial Equivalents = Yes—to turn parallel execution off. In addition, if you want to compile files in which you added Intel Cilk Plus keywords and reducer hyper-objects using another compiler that does not support them, add the following code shown in figure 3 to the top of the file in which the keywords are used:

```

#ifndef __cilk
#include <cilk/cilk_stub.h>
#endif
#include <cilk/cilk.h>

```

Figure 3

The *cilk_stub.h* header file will essentially comment out the keywords so that other compilers will compile the files without any further source code changes. See the “Intel® Cilk™ Plus” section of the Intel C++ Compiler 13.0 User and Reference Guide and other samples in the Intel Cilk Plus sample directory, and other Intel Parallel Studio XE samples, to learn about reducer hyper-objects. These are simple, powerful objects used to protect shared variables among Intel Cilk Plus tasks, without the need for barriers or synchronization code. Use the above instructions to enable the serial mode and rebuild and compare the execution performance of the serial code with the performance of the parallel code you recorded earlier.

Performance with Intel Cilk Plus Array Notation and `cilk_for` - MonteCarlo Simulation Example

Now we will look at a more complex example – a financial Monte Carlo Simulation. This example combines the array notation and the keywords of Intel Cilk Plus to give you both parallelization of the main driver loop using `cilk_for`, and the array notation for the simulation kernel to allow vectorization.

Array notation provides a way to operate on slices of arrays using a syntax the compiler understands and subsequently optimizes, vectorizes, and in some cases parallelizes. This is the basic syntax:

```
[<lower bound> : <length> : <stride>]
```

where the *<lower bound>*, *<length>*, and *<stride>* are optional, and have integer types. The array declarations themselves are unchanged from C and C++ array-definition syntax. Figure 4 shows some example array operations and assignments:

```
Operations:
a[:] * b[:] // element-wise multiplication
a[3:2][3:2] + b[5:2][5:2] // matrix addition of 2x2 subarrays within a and b starting at
a[3][3] and b[5][5]
a[0:4][1:2] + b[0][1] // adds a scalar b[0][1] to each element of the array section in a

Assignments:
a[:, :] = b[:, :][2] + c;
e[:] = d; // scalar variable d is broadcast to all elements of array e
```

Figure 4

Now, take a look at the MonteCarloSample application.

1. Load the MonteCarloSample solution into Microsoft Visual Studio
2. Open the file, mc01.c and look at the function, Pathcalc_Portfolio_Scalar_Kernel. In the scalar kernel function, the scalar array declarations are as follows:

```
__declspec(align(64)) float B[nmat], S[nmat], L[n];
```

In order to utilize array operations, we are going to change the operation of the function, and work on “stride” elements, instead of working on a single element. We start by changing B, S and L from single dimensional to two dimensional arrays:

```
__declspec(align(64)) float B[nmat][vlen], S[nmat][vlen], L[n][vlen];
```

where we have simply specified the size (nmat or n) and length (vlen). We also have to declare some other arrays in the new function, Pathcalc_Portfolio_Array_Kernel() to handle the many scalar accumulations and assignments in the computation loops within the kernel. The resulting code is very similar to the scalar version except for the array section specifiers. For example, figures 5 and 6 are a comparison of one of the loops in the kernel:

Scalar version:

```
for (j=nmat; j<n; j++) {
    b = b/(1.0+delta*L[j]);
    s = s + delta*b;
    B[j-nmat] = b;
    S[j-nmat] = s;
}
```

Figure 5

A Simple Path to Parallelism with Intel® Cilk™ Plus

Array notation version:

```

for (j=nmat; j<n; j++) {
  b[:] = b[]/(1.0+delta*L[j][:]);
  s[:] = s[] + delta*b[];
  B[j-nmat][:] = b[];
  S[j-nmat][:] = s[];
}

```

Figure 6

With some straightforward changes to the code, we have implemented array operations that work on multiple elements at a time, allowing the compiler to use SIMD code and gain a substantial speedup over the serial, scalar kernel implementation.

- Now, let's parallelize the calling loop using `cilk_for`. Take a look at the function, `Pathcalc_Portfolio_Scalar()` in figure 7. All we need to do is replace the `for` with `cilk_for`. This is done for you in the `Pathcalc_Portfolio_Cilk()` function:

```

void Pathcalc_Portfolio_CilkArray(FPPREC *restrict z,
                                FPPREC *restrict v,
                                FPPREC *restrict L0,
                                FPPREC * restrict lambda)
{
  int  stride = SIMDVLEN, path;
  DWORD startTime = timeGetTime();

  cilk_for (path=0; path<npath; path+=stride) {
    Pathcalc_Portfolio_Array_Kernel(stride,
                                    L0,
                                    &z[path*nmat],
                                    lambda,
                                    &v[path]);
  }

  perf_cilk_array = timeGetTime()-startTime;
}

```

Figure 7

- Rebuild the project and run the program by pressing CTRL-F5.

On the 4 core system described in the table on page 2, adding array notation to enable SIMD data parallelism increased performance by close to a factor of two compared to the scalar version. Adding the `cilk_for` keyword to parallelize the calls to the kernel showed good scaling, and resulted in a further speedup of close to a factor of 4 on the same system. This is an illustration of the power and simplicity of Intel Cilk Plus.

Performance with Intel Cilk Plus Array Notation and *cilk_for* - Ambient Occlusion Example

In this sample, we will show the application of Intel Cilk Plus in Visual Computing. This example combines the array notation and the *cilk_for* keyword of the Intel Cilk Plus to vectorize the render code, and parallelize the work across multiple CPU cores.

1. Open the CilkPlus-AOBench folder
2. Choose VS2008 or VS2010 implementations, double-click “aobench_sdl” folder, double click “aobench_sdl” Microsoft Visual Studio Solution. Microsoft Visual Studio* should start, and the project will be loaded into the environment
3. Open the “Source Files” folder in the Solution Explorer on the left-most pane. Open “parallel.cpp” source file by double-clicking it. (See Figure 8 below).

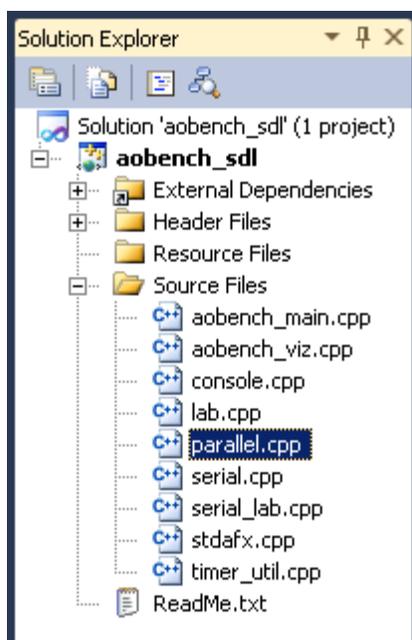


Figure 8

Press Ctrl-F to search, and search for `render_vector_cilk` (figure 9). Hit ESC to close the search window.

```
// Calls the vector code to render one line, and uses Cilk to render multiple lines in parallel.
void render_vector_cilk(float *fimg, int w, int h, int nsubsamples, SDL_Surface *surface, int offset_row, int offset_col)
{
    cilk_for (int y = 0; y < h; y++)
    {
        render_vector_x(fimg, w, h, nsubsamples, y);
        draw_line(surface, offset_row, offset_col, fimg, y, w);
    }
    draw_buffer(surface, offset_row, offset_col, fimg, w, h);
}
```

Figure 9

4. This function has a loop that renders each line of one animation frame (the “y” value is the y-value of the line, and runs from 0 to the height of the frame). We have used the “_Cilk_for” keyword instead of the standard “for” keyword. This indicates that all the loop iterations can be done in parallel using multiple tasks. Each task can be mapped to a different core. Therefore, the simple change from “for” to “_Cilk_for” creates task parallelism, so that each line of the frame can be rendered in parallel across multiple cores.

A Simple Path to Parallelism with Intel® Cilk™ Plus

Now we can examine how the program leverages data parallelism:

- For reference, use the Solution Explorer to open the file “serial.cpp”. Go to the top of the file (press Ctrl-Home) and look at the serial version of the ambient occlusion code, in the function `ambient_occlusion()`. This function renders the individual pixels that make up one line of the image. Don’t worry about how the algorithm computes its result; we are only interested in the differences between the serial and data-parallel versions.
- Go back to “parallel.cpp”, go to the top of the file, and look at the code for the function `ambient_occlusion_arr_notation()` (figure 10). This is the data-parallel version of the first function.

```
int i, j;
int ntheta = NAO_SAMPLES;
int nphi = NAO_SAMPLES;
float eps = 0.0001f;
vec p;
vec basis[3];
float occlusion = 0.0;

// rand1, rand2: buffer arrays to store random results
// theta, phi, x, y, z, rx, ry, rz: array versions of scalar loop variables from serial version
// v: array version of scalar from manually inlined ray_plane_intersect()
// Added the memory alignment explicitly
__declspec(align(64)) float rand1[NAO_SAMPLES], rand2[NAO_SAMPLES], theta[NAO_SAMPLES], phi[NAO_SAMPLES];
__declspec(align(64)) float x[NAO_SAMPLES], y[NAO_SAMPLES], z[NAO_SAMPLES], rx[NAO_SAMPLES], ry[NAO_SAMPLES], rz[NAO_SAMPLES], v[NAO_SAMPLES];
```

Figure 10

You can see that the single variables “x”, “y”, “z”, “theta”, etc. from the serial version have been replaced with arrays: “x[NAO_SAMPLES]”, “y[NAO_SAMPLES]”, etc. We tell the compiler to perform operations on all the array elements at once. Another way of saying this is that instead of working on 1 value at a time, we are working on multiple values at once. NAO_SAMPLES is the number of values we are working with (the vector length). So the basic idea behind using Cilk™ Plus to convert serial code to data-parallel code is to:

- Extend scalar variables to array variables, and
- Use array sections to turn operations on those array variables into data-parallel operations. If you scroll down in the code, you can see many examples of the “[:]” syntax used to create a data-parallel operation. For example:

x[:] = cos(phi[:]) * theta[:];

means, “Take the cosine of all the elements in the array *phi*, multiply the elements with the elements in the array *theta*, and place the results in the array *x*.”

- If you like, you can compile and run all the code yourself by “Ctrl-Alt-F7” to rebuild the project (or by selecting Build Solution under the Build menu), and then executing the demo with the F5 key. The code will run identically to the original version provided.

(For C/C++ Software Developers): Parallelize the code yourself !

- Use the Solution Explorer to open the file “lab.cpp”. This file contains a version of the code that can be parallelized with only a few keystrokes.
- Go to the top of the file (Ctrl-Home), and scroll down about one page to where the variable “is_lab_enabled” is defined. Change the initial value to “true”. Rebuild the application with the F7 key and re-run it with the F5 key.
- The demo has now changed so that it runs a serial version first, and then “your” version, which will start off being identical.
- First, let’s add task-parallelism.
- Go to the bottom of the file with Ctrl-End, and look for the function “render_lab”. There will be a comment directing you to change the “for” to “_Cilk_for”. Please do so.
- What you have done is direct the compiler that each iteration of the for-loop may be done in parallel, and the Cilk Plus runtime will do so when it is productive.
- Build and run the project again. You will see a dramatic improvement in performance.
- For additional performance, you can add data-parallelism.
- Use Ctrl-F to find the string START OF CODE.

A Simple Path to Parallelism with Intel® Cilk™ Plus

10. This brings you to a section of code that is ready to be data-parallelized. This code is a for-loop that runs an index variable k from 0 to `NAO_SAMPLES`. We will remove the loop and change the single-element array operations to data-parallel operations on the whole array.
11. Comment out the for statement. Then go to the end of the loop (where the code says “END OF CODE SECTION” and comment out the closing brace for the loop, where the code says to.
12. Now, press “Ctrl-F” to open the Find dialog. Click “Quick Replace”. Find the string “[k]” and replace it with “[:]”, as in the figure 11 below.

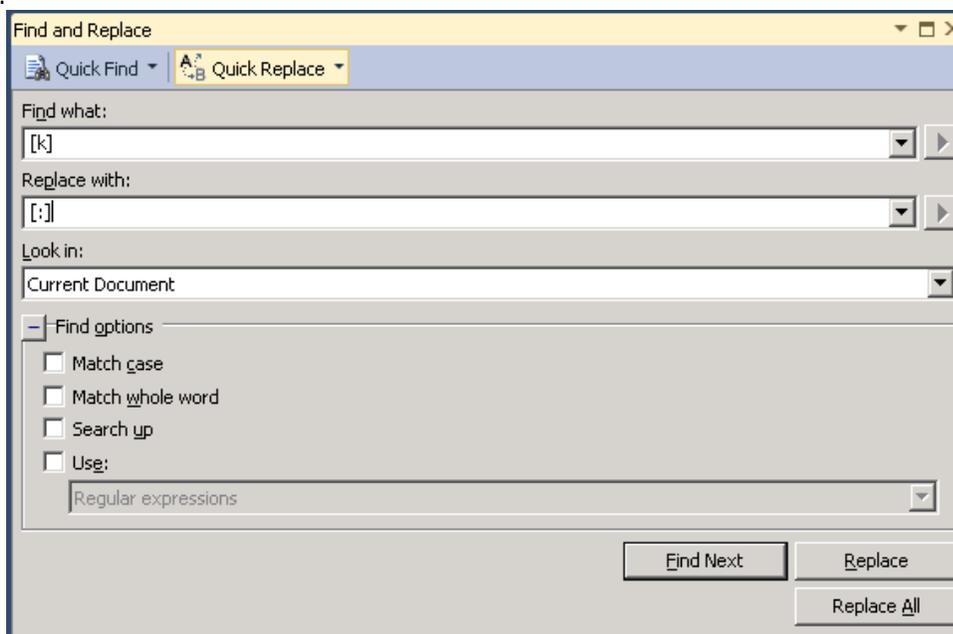


Figure 11

13. Press “Replace All”. You have now replaced all single-element array references such as $x[k]$, $y[k]$, etc. with whole-array references such as $x[:]$, $y[:]$, etc.
14. Build and run the code again. You will see that the data parallelism provides an additional improvement on top of the task parallelism.
15. You’re done! Thanks for exploring the “Many Faces of Parallelism” with Intel® Cilk™ Plus!

Performance with Intel Cilk Plus SIMD Vectorization and Elemental Functions

SIMD Vectorization and Elemental Functions are a part of Intel® Cilk™ Plus feature supported by the Intel® C++ Compiler that provide ways to vectorize loops and user defined functions. The Intel® Compilers provide unique capabilities to enable vectorization. The programmer may be able to help the compiler to vectorize more loops through a simple programming style and by the use of compiler features designed to assist vectorization. Click [here](#) to learn how to use the vector elemental functions, and the SIMD directive (`#pragma simd`) from the Intel® Cilk™ Plus, to help the compiler to vectorize C/C++ code and improve performance.

Summary

Parallelism in a C or C++ application can be simply implemented using the Intel Cilk Plus keywords, reducer hyper-objects, array notation and elemental functions. It allows you to take full advantage of both the SIMD vector capabilities of your processor and the multiple cores, while reducing the effort needed to develop and maintain your parallel code.



Additional Resources

Please visit the user forum for Intel Cilk Plus.

See the Intel Cilk Plus documentation for all of the details about the syntax and semantics:

- [Intel Parallel Studio XE Documentation](#), especially the “Using Intel Cilk Plus” section under “Creating Parallel Applications” in the Intel C++ Compiler 13.0 User and Reference Guide.
- Intel Cilk Plus Tutorial – installed with Intel Parallel Composer and available online along with the Intel Parallel Studio Documentation listed above.

Finally, take a look at our [open specification](#) for Intel Cilk Plus and feel free to comment about it at the email link provided, or on the Cilk Plus user forum listed above.

[Intel Learning Lab](#) – Technical videos, whitepapers, webinar replays and more.

[Intel Parallel Studio XE product page](#) – How to videos, getting started guides, documentation, product details, support and more.

[Evaluation Guide Portal](#) – Additional evaluation guides that show how to use various powerful capabilities.

[Intel® Software Network Forums](#) – A community for developers.

[Intel® Software Products Knowledge Base](#) – Access to information about products and licensing,

[Download a free 30 day evaluation](#)

A Simple Path to Parallelism with Intel® Cilk™ Plus

Standalone product or suite

This evaluation guide focused on the capabilities of Intel® Cilk™ Plus (highlighted in blue below). It comes with the Intel C++ compiler which is available as part of several suites that combine the tools needed to efficiently produce fast, scalable and reliable applications. Single or multi-user licenses along with volume, academic, and student discounts are available.

Suites >>		Intel® Cluster Studio XE	Intel® Parallel Studio XE	Intel® C++ Studio XE	Intel® Fortran Studio XE	Intel® Composer XE	Intel® C++ Composer XE	Intel® Fortran Composer XE
Components	Intel® C / C++ Compiler	●	●	●		●	●	
	Intel® Fortran Compiler	●	●		●	●		●
	Intel® Integrated Performance Primitives ³	●	●	●		●	●	
	Intel® Math Kernel Library ³	●	●	●	●	●	●	●
	Intel® Cilk™ Plus	●	●	●		●	●	
	Intel® Threading Building Blocks	●	●	●		●	●	
	Intel® Inspector XE	●	●	●	●			
	Intel® VTune™ Amplifier XE	●	●	●	●			
	Intel® Advisor XE	●	●	●	●			
	Static Analysis	●	●	●	●			
	Intel® MPI Library	●						
	Intel® Trace Analyzer & Collector	●						
	Rogue Wave IMSL* Library ²							●
	Operating System ¹	W, L	W, L	W, L	W, L	W, L	W, L, O	W, L, O

Note: ¹ Operating System: W=Windows*, L= Linux*, O= OS X*. ² Available in Intel® Visual Fortran Composer XE for Windows with IMSL*

³ Not available individually on OS X, it is included in Intel® C++ & Fortran Composer XE suites for OS X



Learn more about Intel Parallel Studio XE

- Click or enter the link below:
<http://intel.ly/parallel-studio-xe>
- Or scan the QR code on the left



Download a free 30-day evaluation

- Click or enter the link below:
<http://intel.ly/sw-tools-eval>
- Click on 'Product Suites' link

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL 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.

Optimization Notice

Notice revision #20110804

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.