# Intel® Cilk™ Plus Tools User's Guide

This document describes tools designed for use with programs that use the Intel® Cilk™ Plus extensions to the C and C++ languages:

- The Intel® Cilk screen race detector (cilkscreen command)
- The Intel® Cilk view scalability analyzer (cilkview command)

These tools only work with the Intel® Cilk™ portion of the Intel Cilk Plus extensions. They do not currently interact with the C/C++ Extensions for Array Notations (CEAN) portion of Intel Cilk Plus.

## Requirements and Known Limitations

The Intel® Cilk screen (Cilk screen) and Intel® Cilk view (Cilk view) tools require that one of the following be installed:

- Intel® Parallel Composer 2011, Update 1
- Intel® C++ Composer XE 2011, Update 1

These two products include many software development components, including the Intel Cilk Plus run-time system and the Intel® C++ Compiler, which supports the Intel Cilk Plus keywords. For more information about these products, see http://www.intel.com/software/products/.

Attempting to run the Cilk screen and Cilk view tools with an older version of the Intel Cilk Plus run-time system will display a message that no Intel Cilk Plus code was found.

There is a known problem running a 32-bit application under the Cilk screen tool on 64-bit Windows operating systems. This problem is being worked on.

The Cilk view and Cilk screen tools are included in the Intel® Cilk™ Plus SDK, which is available from: http://whatif.intel.com.

# Table of Contents

- The Cilk Screen Race Detector
  - Using Cilk Screen
  - cilkscreen Command Line Options
  - Understanding Cilk Screen Output
  - The Cilk Screen Application Program Interface (API)
    - Disable/Enable Instrumentation
    - Disable/Enable Checking
    - Locks and Fake Locks
    - Cleaning Memory
  - Cilk Screen Performance
- The Cilk View Scalability Analyzer
  - Cilk View Assumptions
  - Using Cilk View
  - cilkview Command Line Options
  - Cilk View Reports
  - Acting on Cilk View Results
  - Cilk View Example
  - Analyzing Portions of an Intel Cilk Plus Program
  - Benchmarking an Intel Cilk Plus Program
  - The Cilk View Application Program Interface (API)
- Legal Information


# The Cilk Screen Race Detector

The Cilk screen race detector (cilkscreen command) monitors the actual operation of an Intel® Cilk™ Plus program as run with some test input. The Cilk screen tool reports all data races encountered during execution. By monitoring program execution, Cilk screen can detect races in your production binary, and can even detect races produced by third-party libraries for which you may not have source code.

To ensure a reliable parallel program, you should review and resolve all races that Cilk screen reports. You can instruct Cilk screen to ignore benign races.

Cilk screen runs your program on a single worker and monitors all memory reads and writes. When the program terminates, it displays information about read/write and write/write conflicts. A race is reported if any possible schedule of the program could produce results different from the serial program execution.

To identify races, run Cilk screen on an Intel Cilk Plus program using appropriate test input data sets, much as you would do to run regression tests. Test cases should be selected so that you have complete code coverage and, ideally, complete path coverage. Cilk screen can only analyze code that is executed, and races may only occur under certain data and execution path combinations.

Please note that Cilk screen only detects and reports races that result from parallelism created by the Intel Cilk Plus keywords. Cilk screen will **NOT** report races that result from conflicts between threads created using other parallel libraries or threads created explicitly using system calls.

Cilk screen recognizes and correctly manages many special cases, including:

- Intel Cilk Plus keywords
- Calls into the Intel Cilk Plus runtime
- Reducers
- Some operating system locks
- Operating system memory management calls (malloc and others)
- Program calls to control Cilk screen

Because Cilk screen understands the parallel structure of your program (that is, which strands can run in parallel and which always run in series), the algorithm is probably correct.

In this chapter, we will explain how to use Cilk screen, explain how to interpret the output, document calls that your program can make to control Cilk screen, and discuss the performance overhead that Cilk screen incurs.

## Using Cilk Screen

To run Cilk screen, either use the command line or the Microsoft Visual Studio* Tools menu.

The cilkscreen command is:

```
cilkscreen [cilkscreen options] [--] your_program [program options]
```

Within Visual Studio*, click the "Tools" menu, and select "Run Cilkscreen Race Detector" from the "Intel Cilk Plus Tools" submenu. Cilk screen runs the program using the command arguments specified in the debugging property of the project.

You can run production or debug versions of your program under Cilk screen. Debug builds typically contain more symbolic information, while release (optimized) builds may include optimizations that make it hard to relate races back to your source code. However, binary code differences may lead to races that are seen only in one version or the other. We recommend testing with debug builds during development, and verifying that production builds are correct before distributing your Intel Cilk Plus program.

To ensure high reliability, we recommend that you run Cilk screen with a variety of input data sets, as different input data may expose different code paths in your program. Because the resolution of one race may expose or create a previously unreported race, you should run Cilk screen after any program changes until you achieve race-free operation. For performance reasons, you may choose to use smaller data sets than you might use for other regression testing. For example, we test our qsort examples for aces using a small array of elements, but run performance tests with large arrays.

Two of the examples include intentional data races. You can build these examples and run them under Cilk screen to see the race reports.

- simple-race, a simple application with two strands that both write to the same global variable.
- qsort-race, one of the Quicksort Examples. The race exists because two spawned recursive function calls use overlapping data ranges.

## Command Line Options

The cilkscreen command recognizes the following options. These options are not available when you run Cilk screen from within Visual Studio.

You can optionally use "--" to separate the name of the program from the options.

Unless specified otherwise, output goes to the stderr (by default, the console).

| Option | Description |
| --- | --- |
| `-r reportfile` | Write output in ASCII format to reportfile. Note that Windows GUI applications do not have stderr, so output will be lost unless you use the -r option. |
| `-x xmlfile` | Write cilkscreen command output in XML format to xmfile. The -x option overrides the -r option; you cannot use both. |
| `-a` | Report all race conditions. The same condition may be reported multiple times. Normally, a race condition will only be reported once. |
| `-d` | Output verbose debugging information, including detailed trace information such as DLL loading. Unless the -l option is specified, the debug output will be written to stderr. |
| `-l logfile` | Write debugging information to logfile. This file is created only if the -d option is specified. |
| `-p [n]` | Pause n seconds (default is one second) before starting the Cilk screen process. |
| `-s` | Display the command passed to PIN. PIN is the dynamic instrumentation package that Cilk screen uses to monitor instructions. |
| `-h, -?` | Display cilkscreen command usage and exit. |
| `-v` | Display version information and exit. |

# Understanding Cilk Screen Output

Consider the following program sum.cpp:

```
01    #include <cilk/cilk.h>
02    #include <iostream>
03
04    int sum = 0;
05
06    void f(int arg)
07    {
08        sum += arg;
09    }
10
11    int main()
12    {
13        cilk_spawn f(1);
14        cilk_spawn f(2);
15
16        cilk_sync;
17        std::cout << "Sum is " << sum << std::endl;
18        return 0;
19    }
```

Build the program with debugging information and run Cilk screen:

| **Windows* OS** | **Linux* OS** |
| --- | --- |
| `icl /Zi sum.cpp` | `icc -g sum.cpp` |
| `Cilkscreen sum.exe` | `Cilkscreen sum` |

On Windows, Cilk screen generates output similar to this:

```
Cilkscreen Race Detector V2.0.0, Build 950 for Intel64

Race condition on location 000000014000A1A0
  write access at 00000001400017B1: (C:\sum.cpp:8, sum.exe!f+0x45)
  read access at 00000001400017AB: (C:\sum.cpp:8, sum.exe!f+0x3f)
    called by 0000000140001916: (C:\sum.cpp:14, sum.exe!main+0x158)
    called by 0000000140004367: (f:\dd\vctools\crt_bld\self_64_amd64\crt\src\crtexe.c:597, sum.exe!__tmainCRTStartup+0x197)
Variable: 000000014000A1A0 - sum

Race condition on location 000000014000A1A0
  write access at 00000001400017B1: (C:\sum.cpp:8, sum.exe!f+0x45)
  write access at 00000001400017B1: (C:\sum.cpp:8, sum.exe!f+0x45)
    called by 0000000140001916: (C:\sum.cpp:14, sum.exe!main+0x158)
    called by 0000000140004367: (f:\dd\vctools\crt_bld\self_64_amd64\crt\src\crtexe.c:597, sum.exe!__tmainCRTStartup+0x197)
Variable: 000000014000A1A0 - sum
Sum is 3
2 errors found by Cilkscreen
Cilkscreen suppressed 1 duplicate error messages
```

Here is what the output means:

## Race condition on location 000000014000A1A0

The tool detected a race condition at memory location 000000014000A1A0.

### write access at 00000001400017B1: (C:\sum.cpp:8, sum.exe!f+0x45)

The first access that participated in the race was a write access from location 00000001400017B1 in the program. This corresponds to source line 8 in sum.cpp, from the instruction at offset 0x45 in the function f() loaded from the executable binary file sum.exe. From the program listing, we see that the source code at line 8 is

```
sum += arg;
```

### read access at 00000001400017AB: (C:\sum.cpp:8, sum.exe!f+0x3f))

The second access was a read from location 00000001400017AB in the program. This also corresponds to source line 8 in sum.cpp, but from the instruction at offset 0x3f in f(). The write access occurs when storing the new value back to sum. The read access occurs when the value of sum is read from memory in order to add the value of arg to it.

### called by 0000000140001916: (C:\sum.cpp:14, sum.exe!main+0x158)

Cilk screen displays a stack trace for the second memory reference involved in the race. It would be very expensive in both space and time to store the stack trace for every memory reference. It is much cheaper to record the stack trace only when a race is detected. Here we see that the caller was at line 14 in sum.cpp, called from offset 0x158 in the function main. We can see in the source code that line 14 is the cilk_spawn:

```
cilk_spawn f(2);
```

The next line of the stack track shows that main was called from the C runtime library function __tmainCRTStartup to start the application.

### Variable: 000000014000A1A0 - sum

The last line in the block shows us the name of the variable involved in the race. This information is not always available.

The next block of output displays the write/write race that was detected on the same memory location. This race occurs between the two attempts to update sum that occur in parallel. The output is very similar to the race we just described.

### Sum is 3

Next we see the output produced by the program itself. Since we didn't direct the diagnostic output to a report file, both appear on the console.

**2 errors found by Cilkscreen**
**Cilkscreen suppressed 1 duplicate error messages**

Finally, Cilk screen summarizes the results. There were two distinct races reported. A third race was a duplicate of one of the reported races, and so the output was suppressed.

# The Cilk Screen Application Program Interface (API)

For the advanced user, Cilk screen provides a set of macros defined in cilkscreen.h that you can use in your program. If you are using the Intel® C++ Compiler (supports C and C++), these macros will generate metadata to implement the requested operation. This mechanism allows you to customize Cilk screen from within your program without measurable cost when you do not run your program under Cilk screen. If you are not using the Intel C++ Compiler, these macros will be compiled away.

## Disable/Enable Instrumentation

Cilk screen begins instrumenting your program when you enter the first *spawning* function; a function that contains a `cilk_spawn` keyword. This instrumentation is expensive, and there are some cases when it could make sense to disable all instrumentation while running part of your program. Enabling instrumentation is very expensive, so these calls should only be used to speed up your program under Cilk screen when you have very large serial sections of code. For example, it might make sense to disable instrumentation if you have a very expensive, serial C++ initialization routine that is called within your spawning function before any parallel constructs are invoked.

```
__cilkscreen_disable_instrumentation();
__cilkscreen_enable_instrumentation();
```

For example:

```
#include <cilk/cilk.h>
#include <cilktools/cilkscreen.h>

int main()
{
    __cilkscreen_disable_instrumentation();
    very_expensive_initialization();
    __cilkscreen_enable_instrumentation();
    cilk_spawn parallel_function1();
    parallel_function2();
    return 0;
}
```

## Disable/Enable Checking

You can turn off race checking without disabling instrumentation. This mechanism is not as safe as a fake lock, but may be useful in limited circumstances. Enabling and disabling checking is much cheaper than enabling instrumentation. However, disabling instrumentation allows your program to run at full speed. With checking disabled, Cilk screen continues to instrument your program with all of the corresponding overhead. While checking is disabled, Cilk screen will not record any memory accesses.

```
__cilkscreen_disable_checking();
__cilkscreen_enable_checking();
```

For example, Cilk screen will not report a race in this program, even though there is in fact a race:

```
#include <cilk/cilk.h>
#include <cilktools/cilkscreen.h>

int sum = 0;
void f(int arg)
{
    sum += arg;
}
int main()
{
    __cilkscreen_disable_checking();
    cilk_spawn f(1);
    cilk_spawn f(2);
    __cilkscreen_enable_checking();
    cilk_sync;
    std::cout << "Sum is " << sum << std::endl;
    return 0;
}
```

Note that enabling and disabling checking is managed with a single counter that starts at 0, is decremented when checking is disabled, and incremented when checking is enabled. Therefore, these calls may be nested; checking is only re-enabled when the counter returns to 0. Do not enable checking if it is not disabled. This is a fatal runtime error and will abort our program.

## Locks and Fake Locks

Cilk screen tracks commonly used locks; critical sections on Windows* and pthread mutexes on Linux* operating systems. If you are using another type of lock, you may need to inform Cilk screen that it has been acquired or released.

```
__cilkscreen_acquire_lock(lock)
__cilkscreen_release_lock(lock)
```

The acquire and release lock macros take the address or handle of your lock.

If you are confident that a race in your program is benign, you can suppress the Cilk screen race report by using a *fake lock*. Simply pass the address of a variable to `__cilkscreen_acquire_lock` and `__cilkscreen_release_lock` and Cilk screen will believe that the region of code is protected by a lock.

Remember that a data race, by definition, only occurs when conflicting accesses to memory are not protected by the same lock. Use a fake lock to pretend to hold and release a lock. This will inhibit race reports with very low run-time cost, as no lock is actually acquired.

Do not forget to call `__cilkscreen_release_lock` in all paths through your code! C++ users may find it helpful to create a small class which calls `__cilkscreen_acquire_lock` when it is constructed, and `__cilkscreen_release_lock` when it goes out of scope and is destructed. For

example:

```
class fake_lock
{
    int _lock;

public:
    fake_lock()
    {
        __cilkscreen_acquire_lock(&_lock);
    }

    ~fake_lock()
    {
        __cilkscreen_release_lock(&_lock);
    }
};
```

For example, Cilk screen will not report a race in this program:

```
#include <cilk/cilk.h>
#include <ciltools/cilkscreen.h>

class fake_lock
{
    int _lock;

public:
    fake_lock()
    {
        __cilkscreen_acquire_lock(&_lock);
    }

    ~fake_lock()
    {
        __cilkscreen_release_lock(&_lock);
    }
};

bool isUpdated = false;

void f(int arg)
{
    fake_lock l;
    isUpdated = true;
}

int main()
{
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return 0;
}
```

## fake_mutex and lock_guard

While you can use `__cilkscreen_acquire_lock` and `__cilkscreen_release_lock` to implement your own fake locks, an implementation of the `fake_lock` class is provided with the Cilk Tools distribution, along with a companion class to tie the acquisition and release of the lock to the scope of the guard variable. The following program demonstrates these classes:

```
#include <cilk/cilk.h>
#include <ciltools/fake_mutex.h>
#include <ciltools/lock_guard.h>

// Global variable we'll race on
bool isUpdated = false;

// Lock to mediate access to the global variable
cilkscreen::fake_mutex m;

void f(int arg)
{
    // The lock "m" will be acquired in the lock_guard constructor, and
    // released when g goes out of scope
    cilkscreen::lock_guard<cilkscreen::fake_mutex> g(m);
    isUpdated = true;
}

int main()
{
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return 0;
}
```

You can use `cilkscreen::lock_guard` with your own classes that provide `lock` and `unlock` methods.

## Using TBB locks

Cilk screen will recognize locks implemented by any package that builds locks based on the underlying operating system primitives. In addition, Cilk screen includes support for TBB spinlocks. Note that you must define the TBB_USE_THREADING_TOOLS macro so that TBB will generate the ITTNOTIFY calls that tell Cilk screen about lock acquisition and release. Failure to define this macro will result in errors reporting lock held across strand boundaries.

```
#include <cilk/cilk.h>

// Cause TBB to notify tools about lock usage
#define TBB_USE_THREADING_TOOLS 1

#include <tbb/spin_rw_mutex.h>

tbb::spin_rw_mutex countMutex;

int foo()
```

```
{
   int z=0;
   cilk_for (int i = 0; i < 10; i++)
   {
       tbb::spin_rw_mutex::scoped_lock lock(countMutex);
       z *= i;
   }
   return z;
}

int main()
{
    foo();
    return 0;
}
```

### Cleaning Memory

Cilk screen needs to recognize when memory is allocated and freed. Otherwise, Cilk screen would report a race if you allocate a memory block, access it, free it, and then allocate another block of memory at the same address. The early accesses would appear to race with the later accesses to the same addresses, even though the memory is logically fresh.

Cilk screen recognizes memory allocation routines provided by standard system libraries, operating system calls, and Miser. If you have routines in your program that act like memory allocators (for example, look-aside lists, sub-allocators, or memory pools) then you can inform Cilk screen when memory should be considered "clean" and thus any previous accesses to that memory should be forgotten.

```
__cilkscreen_clean(void *begin, void *end);
```

Typically, your memory allocator will call `__cilkscreen_clean` just before returning a pointer to freshly allocated memory. For example:

```
void * myAllocator(size_t bytesRequested)
{
    // find memory from our local pool
    void *ptr = getPooledMemory(bytesRequested);
    if (ptr != NULL)
    {
        __cilkscreen_clean(ptr, (char *)ptr + bytesRequested);
    }
    return ptr;
}
```

## Cilk Screen Performance

You will notice that Cilk screen requires significantly more time and memory compared to a regular, unmonitored run of the program.

To keep track of all memory references, Cilk screen uses approximately 5 times as much memory as the program normally need (6x for 64-bit programs.) Thus, a 32-bit program that uses 100 megabytes of memory will require a total of about 600 megabytes when run under Cilk screen.

In addition, when you run your program under Cilk screen, it will slow down by a factor of about 20-40x or more. There are several factors that lead to the slowdown:

- Cilk screen monitors all memory reads and writes at the machine instruction level, adding significant time to the execution of each instruction.
- Programs that use many locks require more processing.
- Programs with many races will run more slowly.
- Cilk screen runs the program on a single worker, so there is no multicore speedup.
- Cilk screen forces the cilk_for grainsize to 1 in order to ensure that all races between iterations are detected.
- Cilk screen runs the program is if a steal occurs at every spawn. This will cause an identity view to be created whenever a reducer is first accessed after a spawn, and will cause the reduction operation to be called at every sync.

We've said that Cilk screen recognizes various aspects of your program such as memory allocation calls, program calls to Cilk screen, and the Intel Cilk Plus keywords. You may wonder whether the performance of your production program is slowed down in order to support this. In fact, Cilk screen uses a metadata mechanism that records information about the program using symbolic information stored in the executable image, but not normally loaded into memory. Thus, Cilk screen calls add virtually no overhead when your program is not run under Cilk screen.

# The Cilk View Scalability Analyzer

The Cilk view scalability and performance analyzer (cilkview command) is designed to help you understand the parallel performance of your Intel Cilk Plus program. Cilk view can:

- Report parallel statistics about an Intel Cilk Plus program
- Predict how the performance of an Intel Cilk Plus program will scale on multiple processors
- Automatically benchmark an Intel Cilk Plus program on one or more processors
- Present performance and scalability data graphically
- Similar to Cilk screen, Cilk view monitors a binary Intel Cilk Plus application. However, Cilk view does not monitor and record all memory accesses, and so incurs much less run time.

In this chapter, we will illustrate how to use Cilk view with a sample Intel Cilk Plus program, document how to control what data is collected and displayed using both API and command line options, and explain how to interpret the output.

## Cilk View Assumptions

Like Cilk screen, Cilk view monitors your program as it runs on a single worker. To estimate performance on multiple cores, Cilk view makes some assumptions:

- If no explicit grain size is set, Cilk view analyzes cilk_for loops with a granularity of 1. This may lead to misleading results. You can use the cilk_grainsize pragma to force Cilk view to analyze the program using a larger grainsize.
- Because the program runs on a single worker, reducer operations such as creating and destroying

views and the reduction operation are never called. If Cilk view does not indicate problems but your program uses reducers and runs slowly on multiple workers, review the section on Reducer performance considerations.

- Cilk view analyzes a single run of your Intel Cilk Plus program with a specific input data set. Performance will generally vary with different input data.

## Using Cilk View

To run Cilk view, either use the command line or the Microsoft Visual Studio* Tools menu.

Run Cilk view from the command line using the cilkview command:

```
cilkview [options] program [program options]
```

The resulting report looks something like this:

```
Cilkscreen Scalability Analyzer V2.0.0, Build 950
1) Parallelism Profile
   Work :                                  53,924,334 instructions
   Span :                                  16,592,885 instructions
   Burdened span :                         16,751,417 instructions
   Parallelism :                           3.25
   Burdened parallelism :                  3.22
   Number of spawns/syncs:                 100,000
   Average instructions / strand :         179
   Strands along span :                    83
   Average instructions / strand on span : 199,914
   Total number of atomic instructions :   18
2) Speedup Estimate
   2 processors:         1.31 - 2.00
   4 processors:         1.55 - 3.25
   8 processors:         1.70 - 3.25
   16 processors:        1.79 - 3.25
   32 processors:        1.84 - 3.25
```

The exact number will vary depending on the program input, which compiler and runtime is used, and the compiler options selected.

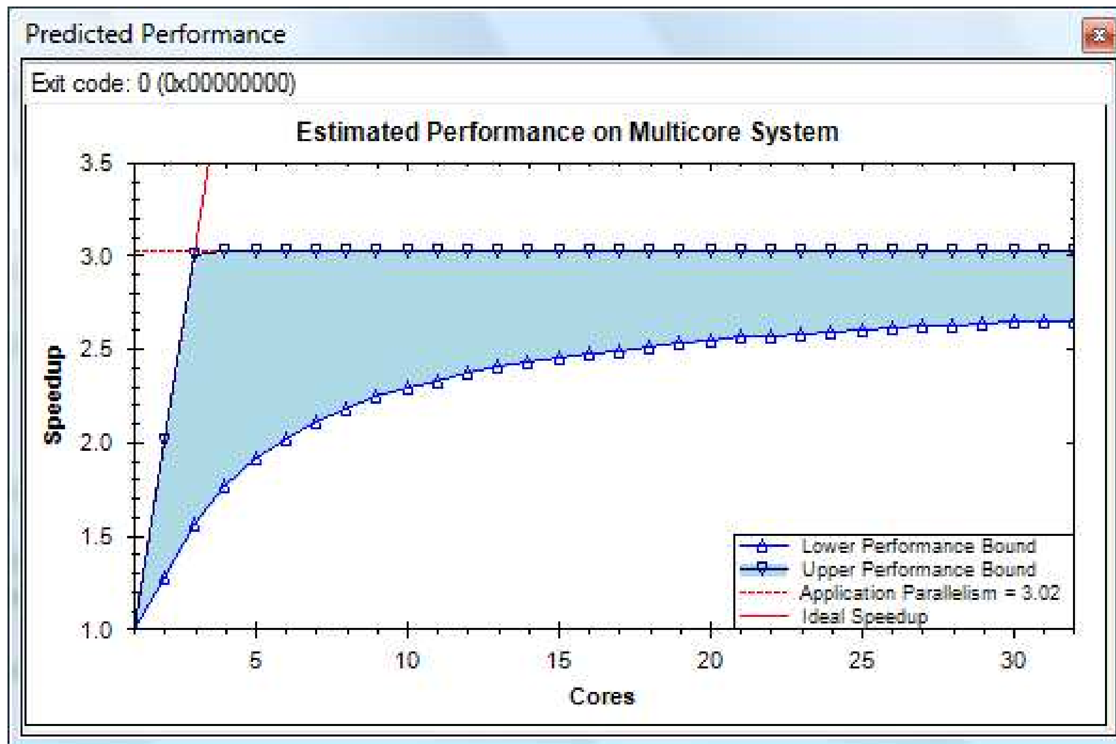
Note: The actual numbers will be different on Linux* and Windows* systems because of compiler differences. Also, the results will vary if the data is shuffled differently, changing the amount of work required by the quick sort algorithm.

The next section describes what the various numbers mean.

On Windows* OS, you can also run Cilk view from within Visual Studio*:

1. Open and build the Intel Cilk Plus project.
2. Select "Tools" -> "Intel Cilk Plus Tools" -> "Run Cilkview Scalability Analyzer" to run the program under Cilk view with the command arguments specified in the project debugging properties.

Cilk view results appear in a Visual Studio window after the program exits and displays a graph that summarizes the parallelism and predicted speedup:




View a copy of the detailed text report by right clicking on the graph and selecting "Show Run Details" from the context menu.

Cilk view writes output into a .csv file. Select "Tools" -> "Intel Cilk Plus Tools" -> "Open Cilkview Log" to view the graph from a previously created .csv log file.

At this time, Cilk view is not fully integrated with Visual Studio. In particular, you cannot specify trial runs using the Visual Studio interface. To collect trial benchmark data, run cilkview from the command line as described above.

## Cilk View Command Line Options

The cilkview command line has the form:

```
cilkview [options] program [program arguments]
```

The cilkview command options are:

| | |
|---|---|
| `-trials none` | Do not run any benchmark trials. This is the default. |
| `-trials one [k]` | Run exactly one trial with k workers (default k = # of cores detected) |
| `-trials log [k]` | Run trials with k, k/2, k/4, .. 1 workers. (default k = # of cores detected) |
| `-trials all [k]` | Run trials with 1..k works. (default k= # of cores detected) |
| `-no-workspan -nw` | Do not generate work and span scalability analysis. |
| `-plot none` | Do not run any plotting program. |
| `-plot gnuplot` | Run gnuplot to display the output. This is the default behavior when Cilk view is run on a Linux* system. |
| `-plot cilkplot` | Run cilkplot to display the output (Windows* systems only). This is the default behavior when Cilk view is run on a Windows system. Cilk Plot is distributed with the Intel® Cilk Tools. |
| `-append` | Append output of this run to existing data files. By default, each run creates a new set of data files. When displaying the output, cilkview will select the fastest run for each tag and thus display the best performance seen. This helps mitigate the indeterminate effect of measuring performance while other processes may be active on your system. |
| `-quiet` | Do not display banners before each benchmark run. |
| `-verbose` | Display extra information such as the PIN command line. Primarily used for debugging. |
| `-v, -version` | Print version information and exit. |
| `-h, -?, -help` | Display cilkview command usage information and exit. |

# What the Profile Numbers Mean

The Cilk view report is broken into two sections, the Parallelism Profile and the Speedup Estimate.

## Parallelism Profile

The Parallelism Profile displays the statistics collected during a single run of the program. To get a complete picture of the parallel operation of the program, Cilk view runs the program on a single worker while tracking all spawns. If the program specifies a grain size for a cilk_for loop, the analysis is run using that grain size. For cilk_for loops that use the default grain size, the grain size for the analysis is set to 1.

In this section, we will use the term overhead to mean the overhead associated with parallelism: scheduling, stealing and synchronization.

Note that because the program is run on a single worker, new reducer views are never created or merged. Thus, the instruction counts do not include the cost of creating new views, calling the reduction function, and destroying the views.

The following statistics are shown:

| | |
|---|---|
| *Work* | The total number of instructions that were executed. Because the program is run on a single worker, no parallel overhead is included in the work count. |
| *Span* | The number of instructions executed on the critical path. |
| *Burdened Span* | The number of instructions on the critical path when overhead is included. |
| *Parallelism* | The *Work* divided by the *Span*. This is the maximum speedup you would expect if you could run the program on an infinite number of processors with no overhead. |
| *Burdened Parallelism* | The *Work* divided by the *Burdened Span*. |
| *Number of spawns* | The number of spawns counted during the run of the program. |
| *Average instructions / strand* | The *Work* divided by the number of strands. A small number may indicate that the overhead of using cilk_spawn is high. |
| *Strands along span* | The number of strands in the critical path. |
| *Average instructions / strand on span* | The *Span* divided by the *Strands along span*. A small number may indicate that the scheduling overhead will be high. |
| *Total number of atomic instructions* | This correlates with the number of times you acquire and release locks, or use atomic instructions directly. |

### Speedup Estimate

Cilk view estimates the expected speedup on 2, 4, 8, 16 and 32 processor cores. The speedup estimates are displayed as ranges with lower and upper bounds.

- The upper bound is the smaller of the program parallelism and the number of workers.
- The lower bound accounts for estimated overhead. The total overhead depends on a number of factors, including the parallel structure of the program and the number of workers. A lower bound less than 1 indicates that the program may slow down instead of speed up when run on more than one processor.

## Acting on the Profile Results

The numbers in the profile report help you understand the parallel behavior of your program. Here are some things to look for:

## Work and Span

These are the basic attributes of a parallel program. In a program without any parallel constructs, the work and span will be the same. Even an infinite number of workers will never execute the program more quickly than a single worker executing the span. If the two numbers are close together, there is very little computation that can be done in parallel. Look for places to add cilk_spawn and cilk_for keywords.

## Number of spawns and Strands along span

These counters provide some insight into how your program executes.

## Low Parallelism

The parallelism is a key number, as it represents the theoretical speedup limit. In practice, we find that you usually want the parallelism to be at least 5-10 times the number of processors you will have available. If the parallelism is lower, it may be hard for the scheduler to effectively utilize all of the workers. If the parallelism is smaller than the number of processors, additional workers will remain idle.

Low parallelism can result from several factors:

- If the granularity of the program is too low, there is not enough work to do in parallel. In this case, consider using a smaller grain size for loops or a smaller base case for recursive algorithms.
- In some cases, a low parallelism number indicates that the problem solved in the test run is too small. For example, sorting an array of 50 elements will have dramatically lower parallelism that an array of 500,000 elements.
- If only part of your program is written in Intel Cilk Plus, you may have good parallelism in some regions but have limited total speedup because of the amount of serial work that the program does. In this case, you can look for additional parts of the program that are candidates for parallelization.
- Reduce the granularity of cilk_for loops by decreasing the grainsize.

## Burdened Parallelism Lower than Parallelism

Burdened parallelism considers the runtime and scheduling overhead, and for typical programs, is a better estimate of the speedup that is possible in practice. If the burdened parallelism is much lower than the parallelism, the program has too much parallel overhead. Typically, this means that the amount of work spawned is too small to overcome the overhead of spawning.

To take advantage of parallelism, each strand needs to perform more work. There are several possible approaches:

- Combine small tasks into larger tasks.
- Stop recursively spawned functions at larger base cases (for example, increase the recursion threshold that is used in the matrix example.
- Replace spawns of small tasks with serial calls.
- Designate small "leaf" functions as C++ rather than Intel Cilk Plus functions, reducing calling overhead.
- Increase the granularity of cilk_for loops by increasing the grainsize.

### Average instructions per strand

This count is the average number of instructions executed between parallel control points. A small number may indicate that the overhead of using cilk_spawn is high relative to the amount of work done by the spawned function. This will typically also be reflected in the burdened parallelism number.

### Average instructions per strand on span

This count is the average number of instructions per strand on the critical path. A small number suggests that the program has a granularity problem, as explained above.

### Total number of atomic instructions

This counter is the number of atomic instructions executed by the program. This is correlated with the number of locks that are acquired, including those used in libraries and the locks implied by the direct use of atomic instructions via compiler intrinsics. If this number is high, lock contention may become a significant issue for your program.

## Cilk View Example

In this section, we present a sample Intel Cilk Plus program that has a parallel performance issue, and show how Cilk view can help identify the problem.

This program uses cilk_for to perform operations over an array of elements in parallel:

```
static const int COUNT = 4;
static const int ITERATION = 1000000;
long   arr[COUNT];
 . . .
long do_work(long k)
{
    long x = 15;
    static const int nn = 87;

    for(long i = 1; i < nn; ++i)
    {
        x = x / i + k % i;
    }
    return x;
}

void repeat_work()
{
    for (int j = 0; j < ITERATION; j++)
    {
        cilk_for (int i = 0; i < COUNT; i++)
        {
            arr[i] += do_work( j * i + i + j);
        }
    }
}
```

```
int cilk_main(int argc, char* argv[])
{
    . . .
    repeat_work();
    . . .
}
```

This program exhibits speedup of less than 1, also known as slowdown. Running this program on 4 processors takes about 3 times longer than serial execution.

Here is the relevant Cilk view output:

```
1) Parallelism Profile
   . . . . . .
   Span:                          2,281,596,648 instructions
   Burdened span:                 32,281,638,648 instructions
   Parallelism:                   3.06
   Burdened parallelism:          0.22
   . . . . . .
   Average instructions / strand:          698
   Strands along span:                     5,000,006
   Average instructions / strand on span:  456

2) Speedup Estimation
   2 processors:     0.45 - 2.00
   4 processors:     0.26 - 3.06
   8 processors:     0.24 - 3.06
   16 processors:    0.23 - 3.06
   32 processors:    0.22 - 3.06
```

Even though the program has a parallelism measure of 3.06, the burdened parallelism is only 0.22. In other words, the overhead of running the code in parallel could eliminate the benefit obtained from parallel execution. The average instruction count per strand is less than 700 instructions. The cost of stealing can exceed that value. The tiny strand length is the problem.

Looking at the source code, we can see that the cilk_for loop is only iterating 4 times, and the amount of work per iterations is very small, as we expected from the profile statistics.

The theoretical parallelism of the program is more than 3. In principle, the 4 iterations could be split between 4 processors. However, the scheduling overhead to steal and synchronize in order to execute only a few hundred instructions overwhelms any parallel benefit.

In this program, we cannot simply convert the outer loop (over j) to a cilk_for loop because this will cause a race on arr[i]. Instead, a simple fix is to invert the loop by bringing the loop over i to the outside:

```
void repeat_work_revised()
{
    cilk_for (int i = 0; i < COUNT; i++)
    {
        for (int j = 0; j < ITERATION; j++)
        {
            arr[i] += do_work( j * i + i + j);
        }
    }
}
```

Here is the Cilk view report for the revised code:

```
1) Parallelism Profile
   . . . . . .
   Span:                            1,359,597,788 instructions
   Burdened span:                   1,359,669,788 instructions
   Parallelism:                     4.00
   Burdened parallelism:            4.00
   . . . . . .
   Average instructions / strand:         258,885,669
   Strands along span:                    11
   Average instructions / strand on span:  123,599,798
```

The revised program achieves almost perfect linear speedup on 4 processors. As Cilk view reports, the parallelism and burdened parallelism have both improved to 4.00.

## Analyzing Portions of an Intel Cilk Plus Program

By default, Cilk view reports information about a full run of the program from beginning to end. This information may include a significant amount of purely serial code such as initialization and output. Often you will want to restrict the analysis to the smaller sections of your code where you have introduced parallel constructs.

The Cilk view API allows you to control which portions of your Intel Cilk Plus program are analyzed. The API is accessed through methods on a Cilk view object as defined in cilkview.h.

Here is a simple example that analyzes the whole program. When run under Cilk view, a single graph will be created. If Cilk view is not running, the functions do nothing and there is no significant performance impact on your program.

```
#include <cilk/cilk.h>
#include <cilkview.h>

int main ()
{
    cilkview_data_t data;
    __cilkview_query(d);     // begin analyzing
    do_some_stuff();         // execute some parallel work
    __cilkview_report(&d, NULL, "main_tag", CV_REPORT_WRITE_TO_RESULTS);

   return 1;
}
```

In a slightly more complex example, we analyze two separate parallel regions. Cilk view will generate two graphs, labeled `first_tag` and `second_tag`.

```
    cilkview_data_t data;
    __cilkview_query(d);     // begin analyzing
    do_first_stuff();         // execute some parallel work
    __cilkview_report(&d, NULL, "first_tag", CV_REPORT_WRITE_TO_RESULTS);

    __cilkview_query(d);     // begin analyzing second parallel region
    do_different_stuff();    // execute more parallel work
    __cilkview_report(&d, NULL, "second_tag", CV_REPORT_WRITE_TO_RESULTS);
```

You can also use the same tag repeatedly. In this case, Cilk view treats the data as independent runs of the same region, and chooses the run that finishes in the least time for the graph. The following code will analyze the fastest run of `do_parallel_stuff()`.

```
for (int count=0; count<MAX; ++count)
{
    cilkview_data_t data;
    __cilkview_query(d);
    do_parallel_stuff();
    __cilkview_report(&d, NULL, "stuff_tag", CV_REPORT_WRITE_TO_RESULTS);
}
```

## Benchmarking an Intel Cilk Plus Program

In addition to parallel analysis, Cilk view also provides a framework for benchmarking an Intel Cilk Plus program on various numbers of processors. A benchmark run is a complete run of your Intel Cilk Plus program using a specified number of workers. Cilk view measures the total elapsed time, and plots the results. If analysis is enabled, the benchmark data is overlaid on the estimated scalability graph.

To enable benchmarking, use the -trials switch. As detailed in the reference section below, you can specify 0, 1, or multiple trial runs.

## The Cilkview Application Program Interface (API)

```
unsigned long long __cilkview_getticks()
```

Returns a counter that can be used for timing applications. The timer resolution is milliseconds. This function will always return valid values, even if the application is not running under Cilk view.

```
void __cilkview_puts(arg)
```

Writes a string to the Cilk view log. Be sure to include any required newlines.

This function does nothing if the application is not running under Cilk view, and will be compiled away if not compiled using the Intel C++ Compiler.

```
void __cilkview_query(cilkview_data_t d)
```

Fetches the current set of Cilk view counters. If successful, d.status will be non-zero.

This function does nothing if the application is not running under Cilk view, and will be compiled away if not compiled using the Intel C++ Compiler.

```
void __cilkview_report(cilkview_data_t *start,
                       cilkview_data_t *end,
                       char *tag,
                       unsigned int flags)
```

Writes information about the interval between the start and end data to either the report, the data file, or both.

| | |
|---|---|
| `start` | A pointer to a `cilkview_data_t` that was previously initialized by a call to `__cilkview_query()`. |
| `end` | A pointer to a `cilkview_data_t` that was previously initialized by a call to `__cilkview_query()`, or NULL. If NULL is passed, Cilk view will use the current values as the end point. |
| `tag` | A string representing the "tag" to be used for this report. There may not be spaces in the tag string. If multiple reports are written with the same tag, Cilk view will plot the smallest one. |
| `flags` | Flags to indicate whether the report is to be written to the log or the results file. Possible values are CV_REPORT_WRITE_TO_LOG and CV_REPORT_WRITE_TO_RESULTS. The values may be ORed together to write the data to both files. |

This function does nothing if the application is not running under Cilk view, and will be compiled away if not compiled using the Intel C++ Compiler.

## View Data with Cilk Plot

The Intel®: Cilk Plot program is provided for Windows systems only. The cilkplot command line has the form:

```
cilkplot TAGNAME.csv
```

where TAGNAME.csv is a data file produced by Cilk view. This allows you to easily view a plot of Cilk view data without re-running the measurements.

## View Data with gnuplot

Cilk view writes TAGNAME.csv and TAGNAME.plt files that can be read by gnuplot. To display the data with gnuplot on Linux or Windows systems run gnuplot (or wgnuplot) as:

```
gnuplot TAGNAME.plt
```

where TAGNAME.plt is the gnuplot script file produced by Cilk view. This allows you to easily view a plot of Cilk view data without re-running the measurements.

# Legal Information

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. 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 go to:
http://www.intel.com/design/literature.htm
Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to:
http://www.intel.com/products/processor_number/
This document contains information on products in the design phase of development.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel Threading Building Blocks, Intel Viiv, Intel vPro, Intel XScale, InTru, the InTru logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, skoool, the skoool logo, Sound Mark, The Journey 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.

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Copyright (C) 2010, Intel Corporation. All rights reserved.