Intel® Cilk++ SDK Programmer's Guide

Document Number:
322581-001US

Legal Information

Contents

Chapter 1

Introduction

Version 1.10 (October 2009).

This programmer's guide describes the Intel® Cilk++ SDK. The software described in this guide is provided under license from Intel Corporation. See the End User License Agreement (EULA) and the Release Notes for license details.

The Intel Cilk++ SDK provides tools, libraries, documentation and samples that enable developers to use the Cilk++ language to add parallelism to new or existing C++ programs. This release of the SDK provides support for building IA-32 architecture programs (32-bit) that run on the Microsoft Windows* Operating System (OS) and IA-32 and Intel 64 architecture programs (32-bit and 64-bit) that run on the Linux OS*.

Most of the information in this guide pertains to all platforms; differences are marked in the text.

Target audience

This programmer's guide is designed for application developers who will use the Intel Cilk++ SDK to improve performance by adding parallelism to new and existing C++ applications. We assume that the reader has a working knowledge of C++ programming. Expert-level knowledge of C++ will be helpful.

Getting started

We recommend that the reader first install the Intel Cilk++ SDK, then build and run at least one of the example programs in order to validate the installation and to begin to get familiar with the compiler and tools. See ***Getting Started*** (Page 8) for details.

The ***Cilk++ Concepts*** (Page 31) and ***Cilk++ Language*** (Page 35) sections provide a good conceptual framework to understand the Cilk++ model of parallelism.

Next, read about ***Race Conditions*** (Page 87), learn how to use the ***Intel Cilk++ cilkscreen race detector*** (Page 96) to identify race bugs, and how ***Reducers*** (Page 52) can be used to eliminate many common race problems.

If you are planning to convert legacy serial applications, read the ***Mixing C++ and Cilk++ Code*** (Page 81) chapter.

Typographic conventions

We use a `monospaced font` for commands, code, keywords and program output.

Pathnames are separated with back slash ("\") for Windows OS and forward slash ("/") for Linux OS. When the environment could be either Windows OS or Linux OS, we use the Linux OS forward slash convention.

TECHNICAL SUPPORT

We want you to be successful using the Intel® Cilk++ SDK. If you have feedback, questions, or problems, please use the support forums at http://whatif.intel.com.

With problem reports, please include the following information:

- Version of the Intel Cilk++ SDK (for example, Cilk++ SDK 1.1.0 for Linux* OS, 64-bit edition, build 7982)
- Operating system and version (for example, Windows Vista* with Service Pack 1, Ubuntu 9.04)
- On Windows systems, please specify the compiler version (for example, Microsoft Visual Studio* 2005 with Service Pack 1)
- A detailed description of the question, problem or suggestion
- Source code whenever possible
- Warning or error messages

RELEASE NOTES

The Release Notes list the system requirements, installation notes, and major changes from the previous release, such as new features, bug fixes, and examples.

The release notes are available online from the http://whatif.intel.com web site.

ADDITIONAL RESOURCES AND INFORMATION

There is a wealth of additional and supplementary information available about the Cilk++ language at http://whatif.intel.com.

The Cilk++ language is based on concepts developed and implemented for the Cilk language at MIT. To learn more about the history of the Cilk language, visit the following links:

- The ***Cilk Implementation Project site*** (http://supertech.csail.mit.edu/cilkImp.html) is a gateway to the MIT Cilk project. A ***project overview*** (http://supertech.csail.mit.edu/cilk/) with links to a set of three lecture notes provides extensive historical, practical, and theoretical background information.
- ***"The Implementation of the Cilk-5 Multithreaded Language"*** (http://supertech.csail.mit.edu/papers/cilk5.pdf) by Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, won the ***Most Influential 1998 PLDI Paper award*** (http://software.intel.com/en-us/articles/Cilk-Wins-Most-Influential-PLDI-Paper-Award) at the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation.

Chapter 2

Getting Started

NOTE: For system requirements and installation instructions, see the Release Notes.

Overview of the Cilk++ language

The Cilk++ language extends C++ to simplify writing parallel applications that efficiently exploit multiple processors.

The Cilk++ language is particularly well suited for, but not limited to, *divide and conquer* algorithms. This strategy solves problems by breaking them into sub-problems (tasks) that can be solved independently, then combining the results. Recursive functions are often used for divide and conquer algorithms, and are well supported by the Cilk++ language.

The tasks may be implemented in separate functions or in iterations of a loop. The Cilk++ keywords identify function calls and loops that can run in parallel. The Intel Cilk++ runtime system schedules these tasks to run efficiently on the available processors. We will use the term *worker* to mean an operating system thread that the Cilk++ scheduler uses to execute a task in a Cilk++ program.

Using the Intel® Cilk++ SDK

In this chapter, we first walk you through the steps of building, running and testing a sample program Cilk++ program.

Next, we describe how to convert a simple C++ program into a Cilk++ program.

After walking through the source code conversion, we show you how to build the program, test it for race conditions, and measure its parallel performance.

When using the Intel Cilk++ SDK, the compiler command names are `cilkpp` on Windows* systems and `cilk++` (or `g++`) on Linux* systems.

BUILD AND RUN A CILK++ EXAMPLE

Each example is installed in an individual folder, as described in the Release Notes. In this section, we will walk through the `qsort` example.

We assume that you have installed the Intel® Cilk++ SDK, and that you have more than one processor core available. If you have a single-core system, you can still build and test the example, but you should not expect to see any performance improvements.

Building qsort

Full, detailed build options are described in the "***Building, Running, and Debugging*** (Page 17)" chapter. For now, use the default settings.

- **Linux* Systems**

    - Change to the qsort directory (e.g. `cd INSTALLDIR/examples/qsort`)
    - Issue the `make` command.
    - The executable `qsort` will be built in the current directory.
    - If `make` fails, check to be sure that the `PATH` environment variable is set to find `cilk++` from `INSTALLDIR/bin`.
- **Windows* Systems**
    - Microsoft Visual Studio* 2005 and 2008 users can open the solution (such as `qsort.sln`) and build the `Release` version. The executable will be built as `EXDIR\qsort\Release\qsort.exe`.
    - From the command line, build `qsort.exe` using the `cilkpp` command: `cilkpp qsort.cilk`
    - If you get an error message, the most likely problem is that the environment is not setup properly. If `cilkpp` fails to find `cl.exe`, be sure that the Visual Studio environment is setup correctly by running the `vcvarsall.bat` script in the `%ProgramFiles%\Visual Studio 8\VC` folder (for Visual Studio* 2005) or `%ProgramFiles%\Visual Studio 9.0\VC` for Visual Studio* 2008. If `cilkpp` is not found, the environment (including path changes) is not setup correctly. Restart the command shell or Visual Studio and try again. If that does not work, you may need to reboot your system.

Running qsort

First, ensure that `qsort` runs correctly. With no arguments, the program will create and sort an array of 10,000,000 integers. For example:

```
>qsort
Sorting 10000000 integers
5.641 seconds
Sort succeeded.
```

By default, a Cilk++ program will query the operating system and use as many cores as it finds. You can control the number of workers using the `cilk_set_worker_count` command line option to any Cilk++ program. This option is intercepted by the Cilk++ runtime system; the Cilk++ program does not see this argument.

Observe speedup on a multicore system

Here are some results on an 8-core system, where the speedup is limited by the application's parallelism and the core count.

```
>qsort -cilk_set_worker_count=1
Sorting 10000000 integers
2.909 seconds
Sort succeeded.

>qsort -cilk_set_worker_count=2
Sorting 10000000 integers
1.468 seconds
Sort succeeded.

>qsort -cilk_set_worker_count 4
```

```
Sorting 10000000 integers
0.798 seconds
Sort succeeded.

>qsort -cilk_set_worker_count 8
Sorting 10000000 integers
0.438 seconds
Sort succeeded.
```

Check for data races

Use the Intel Cilk++ cilkscreen race detector (`cilkscreen`) to verify that there are no data races in the code. Note that any races will be exposed using a small data set. In this example, we sort an array of only 1,000 elements for the race detection analysis. Race conditions are always analyzed with a single-processor run, regardless of the number of processors available or specified. On Windows systems, `cilkscreen` can also be invoked from within Visual Studio*.

```
>cilkscreen qsort 1000   (qsort.exe on Windows systems)
Sorting 1000 integers
0.078 seconds
Sort succeeded.
No errors found by Cilkscreen
```

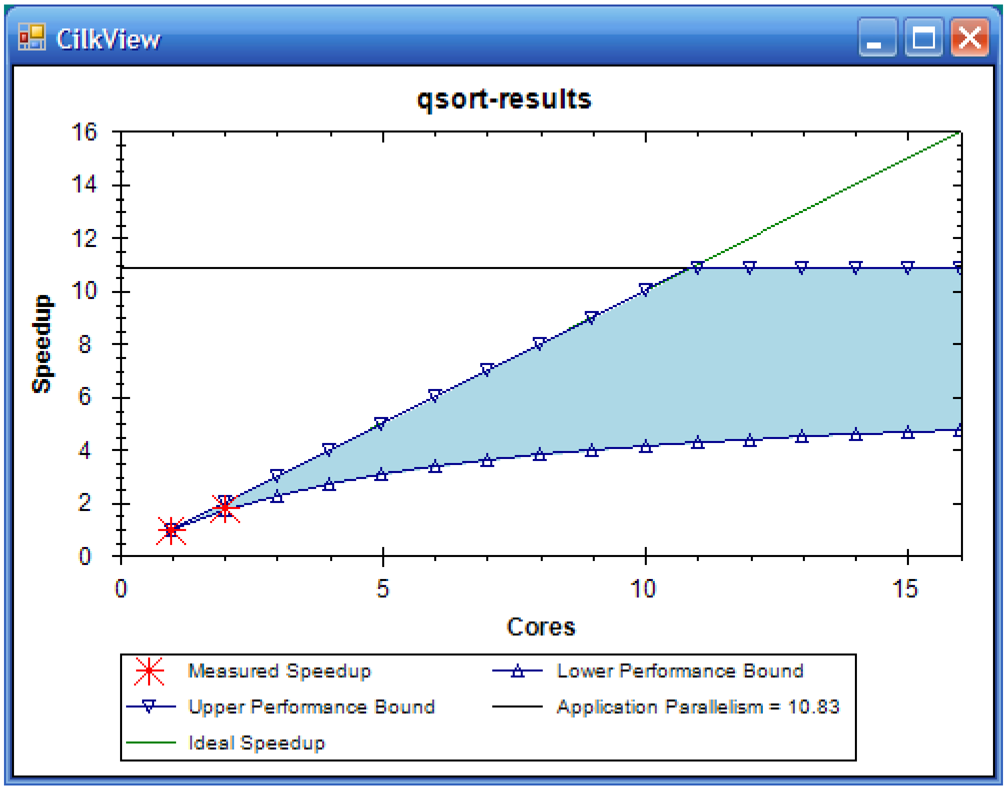
Measure scalability and parallel metrics

Use the Intel Cilk++ cilkview scalability and performance analyzer (`cilkview`) to run your Cilk++ program on multiple processors and plot the speedup. As described in the `cilkview` chapter, the `qsort` example creates a `cilk::cilkview` object and calls the `start()`, `stop()` and `dump()` methods to generate performance measurements. By default, `cilkview` will run the program N times, using 1 to N cores. Use the `-workers` option to specify the maximum number of workers to measure. `cilkview` will run the program one additional time using the Parallel Performance Analyzer option to predict how performance will scale. Here, we run with 1 and 2 workers, plus one additional run. You will see the output of `qsort` each time it runs. After a set of runs, `cilkview` will display a graph showing the measured and predicted performance. The graph, screen and file output are explained in detail in the `cilkview` chapter. (On Linux* systems, the graph is only displayed if `gnuplot` is installed.)

```
>cilkview -trials all 2 -verbose qsort.exe
cilkview: CILK_NPROC=2 qsort.exe
Sorting 10000000 integers
5.125 seconds
Sort succeeded.
cilkview: CILK_NPROC=1 qsort.exe
Sorting 10000000 integers
9.671 seconds
Sort succeeded.
cilkview: cilkscreen -w qsort.exe
Sorting 10000000 integers
38.25 seconds
Sort succeeded.
```

```
Cilkview Scalability Analyzer V1.1.0, Build 7684
1) Parallelism Profile
   Work :                                     17,518,013,236
instructions
   Span :                                     1,617,957,937
instructions
   Burdened span :                            1,618,359,785
instructions
   Parallelism :                              10.83
   Burdened parallelism :                     10.82
   Number of spawns/syncs :                   10,000,000
   Average instructions / strand :            583
   Strands along span :                       95
   Average instructions / strand on span :    17,031,136
   Total number of atomic instructions :      10,000,000

2) Speedup Estimate
   2 processors:          1.73 - 2.00
   4 processors:          2.72 - 4.00
   8 processors:          3.81 - 8.00
   16 processors:         4.77 - 10.83
   32 processors:         5.45 - 10.83
```



CONVERT A C++ PROGRAM

Here is an overview of the sequence of steps to create a parallel program using the Intel® Cilk++ SDK.

- Typically, you will start with a serial C++ program that implements the basic functions or algorithms that you want to parallelize. You will likely be most successful if the serial program is correct to begin with! Any bugs in the serial program will occur in the parallel program, but they will be more difficult to identify and fix.
- Next, identify the program regions that will benefit from parallel operation. Operations that are relatively long-running and which can be performed independently are prime candidates.
- Rename the source files, replacing the `.cpp` extension with `.cilk`.
  - **Windows* OS:** Within Visual Studio*, use the "`Convert to Cilk`" context menu.
- Use the three Cilk++ keywords to identify tasks that can execute in parallel:
  - `cilk_spawn` indicates a call to a function (a "child") that can proceed in parallel with the caller (the "parent").
  - `cilk_sync` indicates that all spawned children must complete before proceeding.
  - `cilk_for` identifies a loop for which all iterations can execute in parallel.
- Build the program:
  - **Windows OS:** Use either the `cilkpp` command-line tool or compile within Visual Studio*.
  - **Linux* OS:** Use the `cilk++` compiler command.
- Run the program. If there are no ***race conditions*** (Page 87), the parallel program will produce the same result as the serial program.
- Even if the parallel and serial program results are the same, there may still be race conditions. Run the program under the ***cilkscreen race detector*** (Page 96) to identify possible race conditions introduced by parallel operations.
- ***Correct any race conditions*** (Page 89) with ***reducers*** (Page 52), locks, or recode to resolve conflicts.
- Note that a traditional debugger can debug the *serialization* (Page 126) of a parallel program, which you can create easily with the Intel Cilk++ SDK.

We will walk through this process in detail using a sort program as an example.

START WITH A SERIAL PROGRAM

We'll demonstrate how to use write a Cilk++ program by parallelizing a simple implementation of ***Quicksort*** (http://en.wikipedia.org/wiki/Quicksort).

Note that the function name `sample_qsort` avoids confusion with the Standard C Library `qsort` function. Some lines in the example are removed here, but line numbers are preserved.

```
 9  #include <algorithm>

11  #include <iostream>
```

```
#include <iterator>
#include <functional>

// Sort the range between begin and end.
// "end" is one past the final element in the range.
// This is pure C++ code before Cilk++ conversion.

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

// A simple test harness
int qmain(int n)
{
    int *a = new int[n];

    for (int i = 0; i < n; ++i)
        a[i] = i;

    std::random_shuffle(a, a + n);
    std::cout << "Sorting " << n << " integers"
              << std::endl;

    sample_qsort(a, a + n);


    // Confirm that a is sorted and that each element
    //   contains the index.
    for (int i = 0; i < n-1; ++i) {
        if ( a[i] >= a[i+1] || a[i] != i ) {
            std::cout << "Sort failed at location i="
                      << i << " a[i] = "
                      << a[i] << " a[i+1] = " << a[i+1]
                      << std::endl;
            delete[] a;
            return 1;
        }
    }
    std::cout << "Sort succeeded." << std::endl;
    delete[] a;
    return 0;
}
```

```
62
63  int main(int argc, char* argv[])
64  {
65      int n = 10*1000*1000;
66      if (argc > 1)
67          n = std::atoi(argv[1]);
68
69      return qmain(n);
70  }
```

CONVERT TO A CILK++ PROGRAM

Converting the C++ code to Cilk++ code is very simple.

- Rename the source file by changing the `.cpp` extension to `.cilk`.
  - **Windows* OS:** Use the "`Convert to Cilk`" context menu within Visual Studio*.
- Add a "`#include <cilk.h>`" statement to the source. `cilk.h` declares all the entry points to the Cilk++ runtime.
- Rename the `main()` function (Line 63) to `cilk_main()`. The Cilk+ runtime system will setup the Cilk++ context, then call this entry point instead of `main()`.

The result is a Cilk++ program that has no parallelism yet.

Compile the program to ensure that the Intel® Cilk++ SDK development environment is setup correctly.

Typically, Cilk++ programs are built with optimized code for best performance.

- **Windows command line:** `cilkpp /Ox qsort.cilk`
- **Windows Visual Studio*:** Specify the `Release` configuration
- **Linux* OS:** `cilk++ qsort.cilk -o qsort -O2`

ADD PARALLELISM USING CILK_SPAWN

We are now ready to introduce parallelism into our `qsort` program.

The `cilk_spawn` keyword indicates that a function (the *child*) may be executed in parallel with the code that follows the `cilk_spawn` statement (the *parent*). Note that the keyword *allows* but does not *require* parallel operation. The Cilk++ scheduler will dynamically determine what actually gets executed in parallel when multiple processors are available. The `cilk_sync` statement indicates that the function may not continue until all `cilk_spawn` requests in the same function have completed. `cilk_sync` does not affect parallel strands spawned in other functions.

```
21  void sample_qsort(int * begin, int * end)
22  {
23      if (begin != end) {
24          --end;  // Exclude last element (pivot)
25          int * middle = std::partition(begin, end,
26                         std::bind2nd(std::less<int>(),*end));
28          std::swap(*end, *middle);      // pivot to middle
29          cilk_spawn sample_qsort(begin, middle);
30          sample_qsort(++middle, ++end); // Exclude pivot
31          cilk_sync;
32      }
33  }
```

In line 29, we spawn a recursive invocation of `sample_qsort` that can execute asynchronously. Thus, when we call `sample_qsort` again in line 30, the call at line 29 might not have completed. The `cilk_sync` statement at line 31 indicates that this function will not continue until all `cilk_spawn` requests in the same function have completed.

There is an implicit `cilk_sync` at the end of every function that waits until all tasks spawned in the function have returned, so the `cilk_sync` at line 32 is redundant, but written here for clarity.

The above change implements a typical divide-and-conquer strategy for parallelizing recursive algorithms. At each level of recursion, we have two-way parallelism; the parent strand (line 30) continues executing the current function, while a child strand executes the other recursive call. This recursion can expose quite a lot of parallelism.

BUILD, EXECUTE AND TEST

With these changes, you can now build and execute the Cilk++ version of the `qsort` program. Build and run the program exactly as we did with the previous example:

Linux* OS:

```
cilk++ qsort.cilk -o qsort
```

Windows* Command Line:

```
cilkpp qsort.cilk
```

Windows Visual Studio*:

build the `Release` configuration

Run qsort from the command line

```
>qsort
Sorting 10000000 integers
5.641 seconds
Sort succeeded.
```

By default, a Cilk++ program will query the operating system and use all available cores. You can control the number of workers using the `cilk_set_worker_count` command line option to any Cilk++ program that uses `cilk_main()`.

Observe speedup on a multicore system

Run qsort using one and then two cores:

```
>qsort -cilk_set_worker_count=1
Sorting 10000000 integers
2.909 seconds
Sort succeeded.

>qsort -cilk_set_worker_count=2
Sorting 10000000 integers
1.468 seconds
Sort succeeded.
```

Alternately, run `cilkview` to get a more detailed performance graph:

```
>cilkview qsort
```

Chapter 3

Build, Run, and Debug a Cilk++ Program

This chapter shows how to build, run, and debug Cilk++ programs in three different development environments:

- Linux* command line
- Windows* command line
- Microsoft Visual Studio*

Build options are described for each environment.

Typically, Cilk++ programs are built for performance and will be built with full optimizations. There are two situations where a debug build (`-g` compiler option for Linux OS, and "Debug" configuration for Microsoft Visual Studio projects) is appropriate:

- You want to build and debug the program's ***serialization*** (Page 126), as described in the "***Debugging Cilk++ Programs*** (Page 26)" chapter.
- You want the diagnostic output from the ***cilkscreen race detector*** (Page 96) to report data races. Using a debug build, `cilkscreen` can provide more complete and accurate symbolic information.

BUILD FROM THE LINUX* COMMAND LINE

In general, compile a Cilk++ program just as you would compile a C++ program, but use the `cilk++` or the `g++` command installed with the Intel® Cilk++ SDK. Be sure that your PATH includes the appropriate `cilk/bin` directory.

The only difference between `cilk++` and the version of `g++` in the Intel Cilk++ SDK is that code compiled with `cilk++` will be treated as Cilk++ code regardless of the file suffix (`.cilk`, `.cpp`, or `.c`). Files compiled with `g++` will be treated as Cilk++ code only if they have a `.cilk` suffix.

In general the `cilk++` and `g++` commands recognize the same set of options as the standard `g++,` with additional Cilk++ language options as described in the reference section. The Cilk++ language options generally fall into one of these three categories:

- Enable or disable various warnings
- Define whether to treat code as C++ or Cilk++ code
- Control optimizations

The optimization options (such as `-O2`) have the same effect with `cilk++` as with `g++`. Be sure to set the appropriate optimization level to maximize the performance of the Cilk++ program.

To generate an IA-32 architecture (32-bit) Cilk++ program executable, use the `-m32` command line option. To generate an Intel® 64 (64-bit) executable, use the `-m64` command line option. The default value is the same as the `cilk++` version (32 or 64-bit). The 64-bit version can only run on 64-bit systems.

Link with the static (rather than the dynamic) Cilk++ runtime library using the `g++` command. Specific command-line arguments are necessary, as follows:

```
g++ -Wl,-z,now -lcilkrts_static -lpthread
```

Example: Build and run a program, such as the `reducer` example, with the following commands:

```
cilk++ -O2 -o reducer reducer.cilk
./reducer
```

COMPILER OPTIONS FOR CILK++ FOR LINUX* OS

The Linux* Cilk++ compiler (invoked either as `cilk++` or `g++`) supports the following `-W` and `-f` options. The standard (non-Cilk++) compilers do not support these options.

As with Linux `g++`, there are two forms. `-fx` (or `-Wx`) enables option `x`, and `-fno-x` (and `-Wno-x`) disables the option. The enabling options are:

`-Wcilk-demote`

Warn when a Cilk++ function uses no Cilk++ keywords and could be declared as a C++ function without affecting its own operation. This does not indicate a bug, and the warning is disabled by default.

`-Wcilk-for`

Warn about suspicious constructs in `cilk_for` loops, such as loop conditions with side effects that will be evaluated different numbers of times in `cilk_for` and ordinary `for` loops. This warning is enabled by default.

`-Wcilk-promote`

Warn when a C++ function is converted ("promoted") to a ***Cilk++ function*** (Page 45). This happens when compiling static constructors and destructors and when instantiating templates. This does not indicate a bug, and the warning is disabled by default.

`-Wcilk-scope`

Warn when the Cilk++ scoping rules for labels change the program's behavior. The scope of a label defined inside a `cilk_for` loop does not extend outside the `cilk_for`, and vice versa. This warning is enabled by default.

`-Wcilk-virtual`

Warn when the language linkage of a virtual function does not match the linkage of the base class function, e.g. overriding a C++ function with a Cilk++ function. The derived class method's linkage is changed to match the base class. (This condition is an error with the Windows* compiler.) This warning is enabled by default.

`-Wredundant-sync`

Warn when a `cilk_sync` statement has no effect because the function has not spawned since the last `cilk_sync`. This warning is enabled by default in this release; the default may change in future versions.

`-fcilk`

Allow Cilk++ keywords. This is on by default, even if the program is being compiled as C++. Cilk++ keywords are not available in C++ functions; this option permits a Cilk++ function to be declared in an otherwise C++ source file.

`-fcilk-stub`

Use the `-fcilk-stub` option with `cilk++` to "stub out" Cilk++ features and build the ***serialization*** (Page 126) of a Cilk++ program. See the example at the end of the preceding section.

`-fcilk-check-spawn-queue`

The Intel Cilk++ runtime only permits 1024 outstanding spawns in a single thread. If this option is enabled, the compiler inserts a runtime check to detect exceeding that limit. If this option is disabled, spawn queue overflow may cause memory corruption. This option is enabled by default. The Cilk++ spawn limit, like the system stack limit, is typically exceeded in the case of a program bug that causes infinite recursion.

`-fcilk-demote`

Convert local functions from Cilk++ to C++ when it is safe to do so. This is enabled by default and should not need to be set to disabled.

`-fcilk-hyper-lookup`

Enable reducer lookup optimization.

`-fcilk-optimize-sync`

Eliminate redundant `cilk_sync` statements; e.g., when a block that ends with an implicit sync also contains an explicit sync. This is enabled by default and should not need to be changed.

`-fimplicit-cilk`

Use Cilk++ language linkage by default in a file that would otherwise be considered C++. It is not normally necessary to use this option unless you are running `cc1plus` directly.

`-finline-cilk-alloc`

Attempt to inline Cilk++ function epilogues. This is enabled by default and should not need to be disabled.

`-finline-cilk-epilogue`

Attempt to inline Cilk++ frame allocation. This is enabled by default when optimizing and should not need to be disabled.

`-fno-cilk-demote`

Small leaf Cilk++ functions that do not use the stack do not allocate a stack frame, improving performance. This optimization is enabled by default, and may be disabled with this flag.

`-x cilk++`

Treat the following files as a Cilk++ source file regardless of the file suffix. `-x none` on the same command line will turn off this option so as to treat subsequent files normally according to their suffixes.

`-v`

As in `gcc`, output version and other information to `stderr`. The information includes the vendor build number, such as:

```
 gcc version 4.2.4 (Cilk Arts build 6252)
```

BUILD FROM THE WINDOWS* COMMAND LINE

To build a Cilk++ program within a Windows* command shell, use the `cilkpp` command installed in the `bin` directory of the Intel® Cilk++ SDK installation.

The `cilkpp` command executes a series of programs that pre-process the Cilk++ program, run the Microsoft* compiler, post-process the compiler output, and finally link the program with the appropriate Intel Cilk++ runtime libraries.

The `cilkpp` program accepts and passes most `cl` options through to the Microsoft* compiler, including those for code optimization. In general, use the same options as you would use for a C++ program. `cilkpp` will issue a warning for any options that the Cilk++ compiler does not support, as enumerated in the `cilkpp` reference section.

CILKPP OPTIONS

cilkpp options

The `cilkpp` program invokes a series of programs in order to build a Cilk++ program.

`cilkpp` supports the following options:

`/cilkp cpp`

Compile the code as C++, removing all `cilkpp`-specific command options from the command line and option files before passing the command to the Microsoft* `cl` compiler to produce the Cilk++ program *serialization* (Page 126). This option forces the inclusion of the `cilk_stub.h` file (see the end of this section).

`/cilkp keep`

Keep intermediate files. `cilkpp` normally deletes these files when they are no longer needed.

`/cilkp serial-debug`

This will turn off parallelism within a single source file and within everything that is called from that source file. The source file compiled with this option will use the native MSVC++ frame layout, so the debugger can show variables within that source file. Parallelism is not suppressed for the rest of the program.

`/cilkp skippreproc`

Skip the preprocessor step used to generate `.tlh` and `.tli` files, which the `cl` compiler generates when it sees `#using` directives. If your code doesn't have `#using` directives, this option can shorten build time.

`/cilkp verbose`

Display internal debugging information as `cilkpp` runs. Intended for vendor debugging use.

`/cilkp version`

This provides the vendor build number and version.

`/cilkp vs2005`

Use only Visual Studio* 2005 tools.

`/cilkp v2008`

Use only Visual Studio 2008 tools.

`/cilkp vs2008Exp`

Use only Visual Studio 2008 Express Edition tools.

`/TK`

Compile files other than `.cilk` files as Cilk++.

`/cilka <option>`

Pass an option to the Cilk++ compiler post-processor. Intended for vendor debugging use.

`/cilkf <option>`

Pass an option to the Cilk++ compiler pre-processor. Intended for vendor debugging use.

`/isystem <include directory>`

Add the specified directory to the include search path, and treat this directory as a system include directory. Certain compiler warnings are disabled for header files found in system directories. The result is the same as if a `.sysinclude` file exists in the specified directory. At installation time, the Intel® Cilk++ SDK installer creates `.sysinclude` files in known system directories.

The Cilk++ compiler can also process `.cpp` C++ files with Cilk++ keywords, creating a serialization, but be certain to include `cilk_stub.h` before `cilk.h`. Do this with the `/cilkp cpp` switch on the `cilkpp` command line.

CL options not supported by the Cilk++ compiler

`cilkpp` supports most of the `cl` command line options However, the following `cl` options are NOT supported:

- `/E` - Preprocess to `stdout`
- `/EHa` - Support asynchronous (structured) exceptions
- `/EP` - Preprocess to `stdout`, no line numbers
- `/FA` - Configure assembly listing
- `/Fa` - Name assembly listing
- `/Gd` - Default calling convention is `__cdecl`
- `/GL` - Whole program optimization
- `/Gm` - Minimal rebuild
- `/Gr` - Default calling convention is `__fastcall`
- `/GS` - Security cookies

- `/Gz` - Default calling convention is `__stdcall`
- `/openmp` - Enable ***OpenMP 2.0*** (http://openmp.org/wp/) language extensions
- `/P` - Preprocess to file
- `/Yc` - Create precompiled header file
- `/Yu` - Use precompiled header file
- `/Zg` - Generate function prototypes
- `/Zs` - Syntax check only

BUILD WITHIN MICROSOFT VISUAL STUDIO*

The Intel® Cilk++ SDK includes an *add-in* for Visual Studio that makes it easy to convert C++ programs to Cilk++ and run the compiler, cilkscreen race detector and cilkview scalability and performance analyzer using the Visual Studio Integrated Development Environment (IDE).

Cilk++ language support is integrated into Visual Studio as a variant of C++, not as a complete new language. Therefore, Visual Studio does not offer a Cilk++ project type. The simplest way to create a Cilk++ project is to create and then convert an existing C++ project. For new projects, create a simple C++ shell program, and convert it to Cilk++.

This section shows how to convert an existing C++ project to Cilk++, how to set compiler options, and how to run Cilk++ programs.

Please note that Visual C++ 2008 Express Edition* does not support add-ins. While the Express Edition can build an existing C++ project containing Cilk++ modules (such as the Cilk++ Example projects), it cannot convert a module from C++ to Cilk++ and copy the compiler settings. If you must use the Express Edition, it may be simpler to build from the command line with `cilkpp`.

CONVERT C++ TO CILK++ WITHIN VISUAL STUDIO*

Cilk++ files use the file suffix `.cilk`, causing Visual Studio to use the `cilkpp` program to compile the file. To add a Cilk++ file to an existing C++ project, simply open the New File dialog. Select "`Visual C++`" in the category tree, and then "`Cilk++ File (.cilk)`" in the Templates pane. Then click the "`Open`" button.

There are two options to convert a C++ file to Cilk++. First, select a C++ file in the Solution Explorer and right click on it to bring up the context menu. The Cilk++ language addin provides two new options:

`Build as Cilk`

> Creates a new `.cilk` file which includes the C++ file, copies any file-specific C++ settings for the file to file-specific Cilk++ settings and excludes the C++ file from the build.

`Convert to Cilk`

Renames the C++ file to `.cilk` and copies any file-specific C++ settings to the file-specific Cilk++ settings.

The correct choice depends on whether you need to support other compilers. "`Build as Cilk`" leaves the existing C++ file in place, which facilitates maintaining a multi-platform build.

If this is the first `.cilk` file in the project, the project-wide C++ settings for all configurations will be copied to the project-wide Cilk++ settings. After this point, changes to the C++ settings will not affect the Cilk++ settings.

Caution:

It is possible that the program will not immediately compile because of mismatched calling conventions. If such a mismatch occurs, there will be an error such as this:

```
error: non-Cilk routine cannot call Cilk routine
```

A mismatch in calling convention happens when a C/C++ function calls a Cilk++ function. After the project has been converted, all functions are considered to be Cilk++ functions except those that are explicitly defined otherwise (the `main()` function is always considered C). To demote functions from Cilk++ to C++ explicitly, use `extern "C++"`, as described in ***Calling C++ functions from Cilk++*** (Page 46).

Caution:

If "`Build as Cilk`" and "`Convert to Cilk`" fail to appear on the Solution Explorer context menu, run the "`ResetAddin.bat`" script in the `visualstudio` directory of the Intel® Cilk++ SDK installation. Visual Studio caches UI elements and may not recognize that the Cilk++ language add in has been added to the environment.

In order to run the script, follow these steps (illustrated for Visual Studio* 2005). When the steps are complete, you may need to restart Visual Studio.

- Open the Visual Studio 2005 Command Prompt: **Start** - **All Programs** - **Microsoft Visual Studio 2005** - **Visual Studio Tools** - **Visual Studio 2005 Command Prompt**
- In the command prompt, change the directory to:
  `"C:\Program Files\Intel\Cilk++\visualstudio"`
- Run `ResetAddin.bat`

SET CILK++ COMPILER OPTIONS WITHIN VISUAL STUDIO*

The ***Compiling Cilk++ Programs from the Command Line*** (Page 17) section described the Cilk++ compiler options. These options can also be set from within Visual Studio.

- Open the Project Properties page, either by right clicking on the project in the Solution Explorer or by selecting **Properties** from the **Project** pull-down list.
- Expand the **Cilk++ Compiler** list in the **Configuration Properties** tree on the left.
- Set the options as required.

In particular, set the same code optimization options as would be used in the C++ program. Code optimization has a similar effect on Cilk++ performance as on C++ performance.

To build the ***serialization*** (Page 126) of the Cilk++ program, which is especially useful for ***debugging Cilk++ programs*** (Page 26), expand the **General** list under **Cilk++ Compiler**. Then set **Compile as C++** to **yes**.

ISSUES BUILDING CILK++ PROGRAMS WITHIN VISUAL STUDIO*

The support for the Cilk++ language integrated into Visual Studio is not completely seamless. While most operations should work fine, there are some known problems and limitations:

- Performing a command-line build by invoking `devenv /build` will cause an access violation exception within Visual Studio if the Cilk++ code is already up-to-date. Using `/rebuild` works around the problem.

  Command line builds should use `MSBuild` or `VCBuild` instead of `devenv`. Both `MSBuild` and `VCBuild` successfully build projects with Cilk++ code.
- There is limited support for debugging Cilk++ applications. See ***Debugging Cilk++ Programs*** (Page 26) for details and workarounds.
- The Cilk++ language is incompatible with incremental linking, so after converting a C++ project to Cilk++ in Visual Studio, the "Enable Incremental Linking" option is turned off (`/INCREMENTAL:NO`). This also applies to command line (`cilkpp`) and Visual Studio builds.
- The Visual Studio integration for `cilkscreen` and `cilkview` does not properly handle attempts to redirect the standard input, standard output, or standard error. For example, if you've set the Command Arguments on the Debugging Properties page to

  ```
      foo bar > out.spot
  ```

  instead of redirecting standard out to the file `out.spot`, all four arguments will be passed to the application. The results are unpredictable and will depend on the application being run. Some applications may terminate due to unrecognized arguments. Note that this is a problem in the integration of `cilkscreen` and `cilkview` with Visual Studio. The command line version of these tools handles redirection correctly.

SET WORKER COUNT

By default, the number of worker threads is set to the number of cores on the host system. In most cases, the default value will work well.

You may increase or decrease the number of workers under program control (if you create a cilk::context) or from the command line or environment.

You may want to use fewer workers than the number of processor cores available in order to run tests or to reserve resources for other programs. In some cases, you may want to oversubscribe by creating more workers than the number of available processor cores. This might be useful if you have workers waiting on locks, for example.

Command Line

You can specify the number of worker threads from the command line, rather than use the default value, with the `-cilk_set_worker_count` option (Windows* programs also accept `/cilk_set_worker_count`). `cilk_main()` processes this option and removes it from the command line string that is visible to the application. Note that the processing of the command line `cilk_set_worker_count` option occurs in the code generated to support `cilk_main()`. If you do not use `cilk_main()` in your program, you will need to decide whether to support this option in your command line processing.

The following are all valid:

```
   qsort
   qsort 100000
   qsort 100000 -cilk_set_worker_count 4
   qsort 100000 -cilk_set_worker_count=4
   qsort -cilk_set_worker_count 4 100000
   qsort -cilk_set_worker_count=4 100000
```

Environment

You can also specify the number of worker threads using the environment variable `CILK_NPROC.`

`Windows* OS: set CILK_NPROC=4`

`Linux* OS: CILK_NPROC=4`

Program Control

If you create an explicit `cilk::context`, you can control the number of workers within your Cilk++ program. Note that you cannot use both `cilk_main()` and create your own `cilk::context` in the same program. See the description of the `cilk::context` interface in the ***Runtime System and Libraries*** (Page 75) chapter.

SERIALIZATION

The Cilk++ language is designed to have serial semantics. In other words, every Cilk++ program corresponds to an equivalent C++ program. We call such a C++ program the ***serialization*** (Page 126) of the Cilk++ program. The serialization is particularly useful when ***debugging Cilk++ programs*** (Page 26). For more information on the value and consequences of serial semantics, see ***4 Reasons Why Parallel Programs Should Have Serial Semantics*** (http://software.intel.com/en-us/articles/Four-Reasons-Why-Parallel-Programs-Should-Have-Serial-Semantics).

How to Create a Serialization

The header file `cilk_stub.h` contains macros that redefine the Cilk++ keywords and library calls into an equivalent serial form. Include `cilk_stub.h` before `cilk.h` to build a serialization.

Linux* OS:

The Cilk++ compiler provides command line options to facilitate serialization. There are two equivalent options:

```
-fcilk-stub
-include cilk_stub.h -fno-implicit-cilk
```

For example, to build the serialized `reducer` example, copy `reducer.cilk` to `reducer.cpp` (which will contain Cilk++ keywords) and build, as follows:

```
cp reducer.cilk reducer.cpp
cilk++ -O2 -fcilk-stub -o reducer_serial reducer.cpp
```

Windows* OS:

The Cilk++ compiler supports serialization:

```
cilkpp /cilkp cpp
```

For example, to build the serialized `reducer` example, copy `reducer.cilk` to `reducer.cpp` (which will contain Cilk++ keywords) and build, as follows:

```
copy reducer.cilk reducer.cpp
cilkpp /O2 /cilkp cpp reducer.cpp
```

This will compile the code as C++, removing all cilkpp-specific command options from the command line and option files before passing the command to the Microsoft `cl` compiler to produce the serialization. This option forces the inclusion of the `cilk_stub.h` file.

DEBUGGING STRATEGIES

Debugging a parallel program tends to be more difficult than debugging a serial program. The Intel® Cilk++ SDK is designed to simplify the challenge of parallel debugging as much as possible. In particular, we recommend debugging the serialization first.

Follow these steps to minimize the problem of debugging parallel programs:

- If you are converting an existing C++ program, debug and test the serial version first.
- Once you have a parallel Cilk++ program, test and debug the serialization. Because both the serial base program and the serialization of the Cilk++ program are serial C++ programs, you can use existing serial debug tools and techniques.
- Use the ***cilkscreen race detector*** (Page 96) to identify races. Resolve the races using one or more of the following techniques, as described in "***Data Race Correction Methods*** (Page 89)":
- Restructure the code to remove the race
- Use a Cilk++ reducer
- Use a `cilk::mutex` lock, other lock, or atomic operation

It may be simpler to debug programs built without optimizations. This will turn off inlining, resulting in a more accurate call stack, and the compiler will not attempt to reorder instructions and optimize register usage.

Linux* OS: The `gdb` debugger provided with the Intel Cilk++ SDK understands the Cilk++ cactus stack and can display stack traces for parallel Cilk++ programs.

Windows* OS: The Visual Studio* debugger does not understand the Intel Cilk++ runtime environment. While you can set a breakpoint in a Cilk++ function or method, you cannot examine variables, and the stack may not display properly. In addition, stepping into a Cilk++ function will step into a function called `__cilk_box()`. In a release build, these calls are inlined and optimized out by the compiler.

Chapter 4

Cilk++ Examples

The Intel® Cilk++ SDK includes examples that illustrate how to write a variety of Cilk++ programs. Each example is in its own directory in the `examples` directory, and each example includes the `.cilk` source code, Linux `make` file, and Microsoft Visual Studio* 2005 solution and project files. Several directories contain a `ReadMe.txt` file with additional information.

Feel free to use the example code as a basis for experimentation and development.

General Examples

bzip2

Linux* OS: The `bzip2` example demonstrates how to parallelize a complete application.

cilk-for

The `cilk-for` example demonstrates how to create a simple `for` loop using the `cilk_for` keyword. Each loop iteration can execute in parallel.

hanoi

The `hanoi` example illustrates the use of a list reducer to collect the output while solving the classic ***Towers of Hanoi*** (http://en.wikipedia.org/wiki/Towers_of_hanoi) problem in parallel.

linear-recurrence

This example computes a linear recurrence relation, and the computation is parallelized with a reducer. The example includes the reducer code, which augments the examples in the "***Reducers*** (Page 52)" chapter.

reducer

The `reducer` example demonstrates how to use a reducer to accumulate values in parallel using a simple "sum" reducer.

sum-cilk

The `sum-cilk` example requires one of the ***Cilk++ Reducers*** (Page 52) and, like `matrix-transpose`, is a good platform to experiment with performance tuning.

wc-cilk

The `wc-cilk` ***example*** (http://software.intel.com/en-us/articles/Multicore-enabling-the-Unix-Linux-wc-word-count-utility) demonstrates concurrent file processing with reducers to accumulate statistics.

Matrix Multiplication and Transpose Examples

There are three matrix multiplication examples and one transpose example that illustrate `cilk_for`, loop granularity, memory management, and other methods to tune application performance. The projects contain `ReadMe.txt` files that describe the code in detail and suggest tuning techniques.

matrix

The `matrix` example multiplies two large matrices, using two distinct algorithms (naive-iterative and recursive). Both algorithms run sequentially and parallel versions, and the timing results are displayed. The recursive algorithm is significantly faster and also provides superior parallel speed up.

matrix_multiply

`matrix_multiply` uses a straight-forward algorithm with three loops to multiply square matrices.

matrix_multiply_dc_notemp

`matrix_multiply_dc_notemp` uses a recursive divide-and-conquer algorithm to multiply square matrices. This solution does not use temporary storage.

matrix-transpose

The `matrix-transpose` example transposes a large square matrix (the size is a command line parameter). The example is set up to allow for a variety of performance tuning experiments by adjusting the `cilk_for` ***grain size***, loop order, and more.

Quicksort Examples

The collection of quicksort examples include several ways to structure a Cilk++ program (including a dynamic link library and a Windows* GUI application), illustrate how to convert a serial C++ program, use a mutex, and find and resolve a race condition.

qsort

The `qsort` example demonstrates how to speed up a ***Quicksort*** (http://en.wikipedia.org/wiki/Quicksort) algorithm by adding Cilk++ keywords. This is essentially the same code as in the "***Getting Started*** (Page 8)" chapter.

qsort-cpp

The `qsort-cpp` example is a C++ program that can be converted to Cilk++. The code is essentially the same code as in the "Converting to Cilk++" section.

qsort-race

The `qsort-race` example includes an intentional race condition. When run under `cilkscreen`, the race is detected and reported to the user.

qsort-mutex

The `qsort-mutex` example demonstrates how to use the Cilk++ mutex library.

qsort-dll

Windows: `qsort-dll` is a Visual Studio* solution with three projects. It shows how to convert a Windows DLL (Dynamic Link Library) from C++ to Cilk++ (refer to *Converting Windows DLLs to Cilk*++ (Page 71) for more details). The projects are `qsort-client`, `qsort-cpp-dll` (a C++ DLL), and `qsort-cilk-dll` (a Cilk++ DLL). `qsort-client` is linked against both DLLs and will call one or the other based on the command line option (`-cpp` or `-cilk`). The Cilk++ DLL will be faster on a multicore system.

QuickDemo

Windows: This example has a GUI interface built with Microsoft Foundation Classes (MFC). Threads that use MFC should not be converted to Cilk++ because of thread-local storage issues. `QuickDemo` demonstrates how to break an application into a UI thread which uses MFC and worker threads that use `SendMessage()` to communicate with the UI thread.

Chapter 5

Cilk++ Concepts

Cilk++ programming is a bit of a departure from traditional serial programming, and requires a somewhat different "world view" in order to write Cilk++ programs that perform and scale well.

In this section, we will introduce and explain some concepts that are fundamental to Cilk++ programming, and indeed, important for any parallel programmer to understand.

First, we will introduce a way to describe the structure of a Cilk++ program as a graph of *strands* and *knots.*

Next, we will discuss how to analyze the expected performance of an Cilk++ program in terms of *work*, *span*, and *parallelism*.

STRANDS AND KNOTS

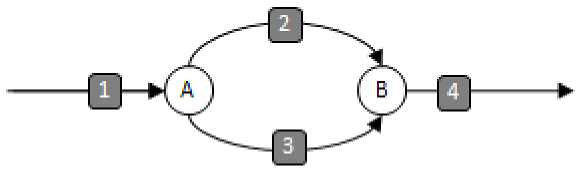
Traditional serial programs are often described using call graphs or class hierarchies. Parallel programming adds another layer on top of the serial analysis. In order to diagram, understand and analyze the parallel performance of a Cilk++ program, we will distinguish only between sections of code that run serially, and sections that may run in parallel.

We will use the word *strand* to describe a serial section of the program. More precisely, we define a strand as "any sequence of instructions without any parallel control structures."

Note that according to this definition, a serial program could be made up of many sequential strands as short as a single instruction each, a single strand consisting of the entire program, or any other partitioning. We will assume that sequential strands are always combined to make a single, longer strand.

In addition, we will define a *knot* as the point at which three or more strands meet. A Cilk++ program will have two kinds of knots - a *spawn knot* and a *sync* knot. Here's a picture illustrating 4 strands (1, 2, 3, 4), a spawn knot (A) and a sync knot (B).

Here, only strands (2) and (3) may execute in parallel.



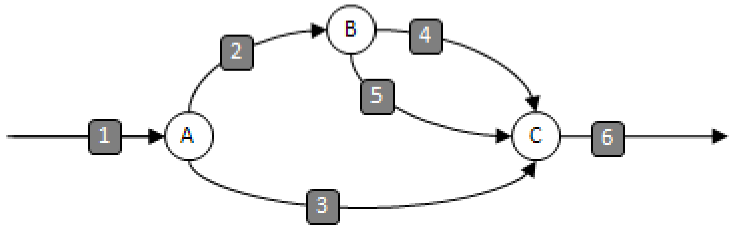
A Cilk++ program fragment that has this structure is:

```
    ...
    do_stuff_1();            // execute strand 1
    cilk_spawn func_3();     // spawn strand 3 at knot A
    do_stuff_2();            // execute strand 2
    cilk_sync;                // sync at knot B
    do_stuff_4();             // execute strand 4
    ...
```

In these illustration, the strands are represented by lines and arcs, while the knots are represented by the circular nodes. We will refer to a strand/knot diagram as a Directed Acyclic Graph (DAG) that represents the serial/parallel structure of a Cilk++ program.

Note: In some published literature (including some papers about Cilk and Cilk++), you will see similar diagrams in which the work is done in the nodes rather than the arcs.

In a Cilk++ program, a *spawn knot* has exactly one input strand and two output strands. A *sync knot* has two or more input strands and exactly one output strand. Here is a DAG with two spawns (labeled A and B) and one sync (labeled C). In this program, the strands labeled (2) and (3) may execute in parallel, while strands (3), (4), and (5) may execute in parallel.



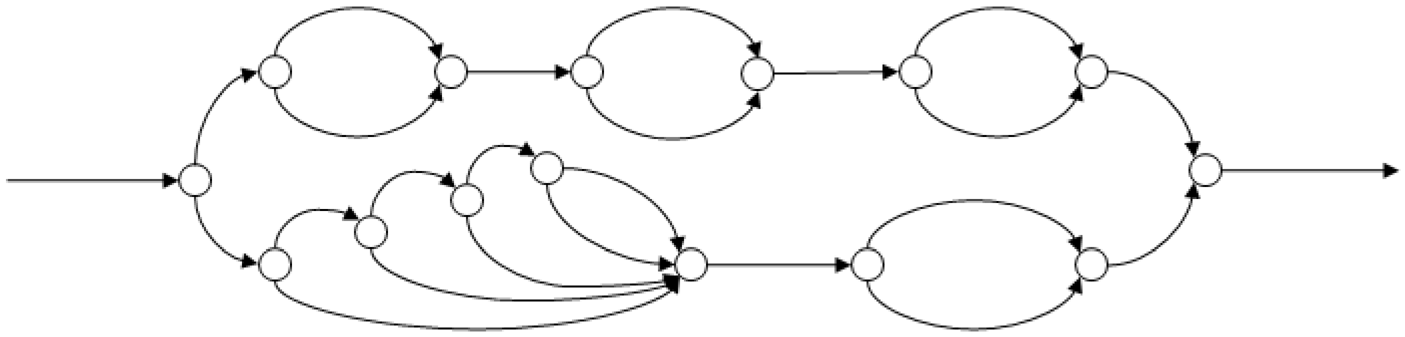
A DAG represents the serial/parallel structure of the execution of a Cilk++ program. With different input, the same Cilk++ program may have a different DAG. For example, a spawn may execute conditionally.

However, the DAG does NOT depend on the number of processors on which the program runs, and in fact the DAG can be determined by running the Cilk++ program on a single processor. Later, we will describe the execution model, and explain how work is divided among the number of available processors.

WORK AND SPAN

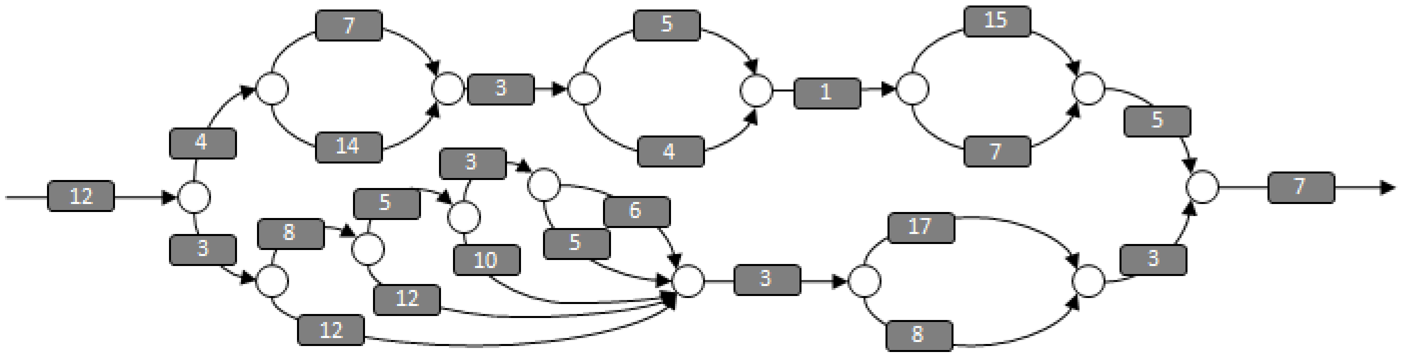
Now that we have a way of describing the serial/parallel structure of a Cilk++ program, we can begin analyze the performance and scalability.

Consider a more complex Cilk++ program, represented in the following diagram.



This DAG represents the parallel structure of some Cilk++ program. The ambitious reader might like to try to construct a Cilk++ program that has this DAG.

Let's add labels to the strands to indicate the number of milliseconds it takes to execute each strand:



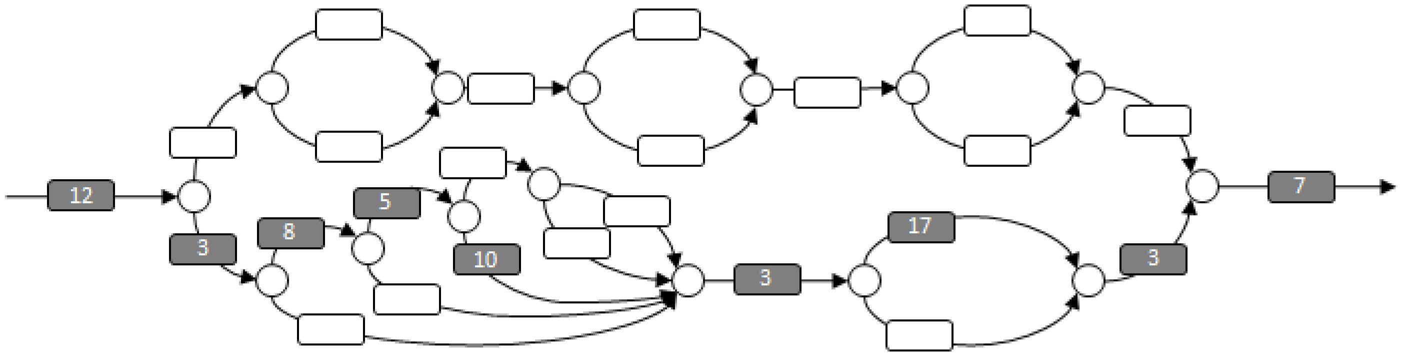
Work

The total amount of processor time required to complete the program is the sum of all the numbers. We call this the *work*.

In this DAG, the work is 181 milliseconds for the 25 strands shown, and if the program is run on a single processor, the program should run for 181 milliseconds.

Span

Another useful concept is the *span*, sometimes called the *critical path length*. The span is the *most expensive* path that goes from the beginning to the end of the program. In this DAG, the span is 68 milliseconds, as shown below:



In ideal circumstances (e.g., if there is no scheduling overhead) then, if an unlimited number of processors are available, this program should run for 68 milliseconds.

With these definitions, we can use the work and span to predict how a Cilk++ program will speedup and scale on multiple processors. The math is fairly simple, but we'll change the names a bit to confuse you.

When analyzing a Cilk++ program, we like to talk about the running time of the program on various numbers of processors. We'll use the following notation:

T(P) is the execution time of the program on P processors.

Thus, using the descriptions of Work and Span:

T(1) is the Work

T(∞) is the Span

Note that on 2 processors, the execution time can never be less than T(1) / 2. In general, we can state the Work Law:

T(P) >= T(1) / P

Similarly, for P processors, the execution time is never less than the execution time on an infinite number of processorrs, hence the Span Law:

T(P) >= T(∞)

Speedup and Parallelism

Intuitively, if a program runs twice as fast on 2 processors, then the speedup is 2. We formalize this by defining the *speedup* on P processors as:

T(1) / T(P)

The maximum possible speedup would occur using an infinite number of processors. Thus, we define the *parallelism* as:

T(1) / T(∞)

Estimating performance and scalability

So what good is all this? Well, if we had some way to measure T(1) and T(∞), then we could predict how much speedup we would expect on P processors, and estimate how well the program scales - that is, the maximum number of processors that might continue to yield a performance improvement.

This is what `cilkview` does. Measuring the work T(1) is of course easy - simply run the program on one processor. If we had a machine with an infinite number of processors available, then it would be easy to directly measure T(∞). Unfortunately, those are hard to come by.

For the rest of us, `cilkview` reports the parallelism by combining its knowledge of the DAG with measurements of the time spent executing each strand. Using these and other measurements, `cilkview` provides a speedup estimate and other information that provides valuable insights into the behavior of a Cilk++ program.

Chapter 6

The Cilk++ Language

Cilk++ is a new language based on C++. This chapter describes how to enter into a Cilk++ context, the Cilk++ keywords - `cilk_spawn`, `cilk_sync` and `cilk_for,` calling between C++ and Cilk++ code, exception handling, and predefined macros that are available.

Note that this version of the Cilk++ language does not support speculative parallelism. There is no mechanism to abort parallel computation.

This chapter does not cover other Intel® Cilk++ SDK components such as the ***Runtime System and Libraries*** (Page 75), the ***cilkscreen race detector*** (Page 96) and other tools, or ***reducers*** (Page 52).

ENTERING A CILK++ CONTEXT

Every program begins as a single-threaded serial program. In order to begin parallel operation, you must create and initialize the Cilk++ runtime *context*, and begin running at a well-defined entry point in your program. You can replace `main()` with `cilk_main()` to use the automatically generated context, or create an explicit context and call `cilk::run` to enter it. The `cilk_main` approach is simpler, but the explicit context provides more control.

cilk_main

In most of the examples, we take advantage of the scaffolding provided by the Cilk++ runtime system by declaring main program entry point to be `cilk_main().` The runtime system automatically creates a set of worker threads, then calls your `cilk_main` entry point in a Cilk++ environment. This approach is very simple to use and works well for new applications that will be written in Cilk++ throughout. A complete description of `cilk_main` is provided in the following section.

Cilk++ Context and cilk::run

Sometimes you may prefer to have a C++ main program, and call into a Cilk++ context explicitly. You might want to use Cilk++ in programs that also use other threading models or libraries such as Microsoft Foundation Classes or programs that create and use pthreads. If you are introducing Cilk++ into a large existing program, you may prefer to convert only a part of the program to Cilk++, and leave other sections in their original serial form. Or, perhaps you are writing a library routine that will be called from other applications that may not be Cilk++ programs.

In these cases, you have the option to create your own Cilk++ context, initialize it explicitly under program control, and call into and return from the Cilk++ context one or more times as needed.

If the program only occasionally requires parallel resources, use the explicit `cilk::run()` interface, which is part of the ***runtime system and libraries*** (Page 75). When you call `cilk::run()`, the workers will all be active until `cilk::run()` returns. At that point, the workers will sleep until the next call to `cilk::run()`. There is some additional cost to entering and leaving a Cilk++ context in this manner, but if sufficient work is performed in the `cilk::run()` environment, that cost is insignificant.

CILK_MAIN

If you replace `main()` with `cilk_main()`, the Cilk++ compiler creates a Cilk++ context at startup time, and calls `cilk_main()` in a parallel environment.

`cilk_main()` takes 0, 2, or 3 arguments with these prototypes:

```
int cilk_main ();
int cilk_main (int argc, char *argv[]);
int cilk_main (int argc, char *argv[], char *env[]);
```

Windows* OS: `cilk_main()` also supports wide characters with two additional prototypes:

```
int cilk_main (int argc, wchar_t *argv[]);
int cilk_main (int argc, wchar_t *argv[], wchar_t *env[]);
```

As a result, you can support generic _tchar characters for parameter types and change a _tmain() function into a cilk_main() without changing any code.

CILK_SPAWN

The `cilk_spawn` keyword modifies a function call statement to tell the Cilk++ runtime system that the function may (but is not required to) run in parallel with the caller. A `cilk_spawn` statement has the following forms:

```
var = cilk_spawn func(args);     // func() returns a value
cilk_spawn func(args);           // func() returns void
```

func is the name of a function which may run in parallel with the current strand. This means that execution of the routine containing the `cilk_spawn` can execute in parallel with *func*. *func* must have Cilk++ linkage, described in the "***Cilk++ and C++ Language Linkage*** (Page 45)" section.

var is a variable with the type returned by *func*. It is known as the ***receiver*** (Page 125) because it receives the function call result. The receiver must be omitted for void functions.

args are the arguments to the function being spawned. Be careful to ensure that pass-by-reference and pass-by-address arguments have life spans that extend at least until the next `cilk_sync` or else the spawned function may outlive the variable and attempt to use it after it has been destroyed. Note that this is an example of a data race which would be caught by `cilkscreen`.

A spawned function is called a ***child*** of the function that spawned it. Conversely, the function that executes the `cilk_spawn` statement is known as the ***parent*** of the spawned function.

Note that a function can be spawned using any expression that is a function. For instance you could use a function pointer or member function pointer, as in:

```
var = cilk_spawn (object.*pointer)(args);
```

SPAWNING A FUNCTION THAT RETURNS A VALUE

If you spawn a function that returns a value, the value should be assigned to a previously declared "receiver" variable before spawning. Otherwise, there will be a compiler error or warning.

Here are three examples, along with the Linux* OS and Windows* OS behavior, and the correct usage.

Example 1: Assign a function value to a variable constructed in the same statement.

```
int x = cilk_spawn f();
```

- **Windows OS**: Won't compile; x is being constructed
- **Linux OS**: Allowed with a warning. `f()` is called, but not spawned. There is no parallelism

The correct form is to declare the receiver variable in a separate statement.

```
int x;
x = cilk_spawn f();
```

Example 2: Spawn the function without a receiver variable.

```
cilk_spawn f();
```

- **Windows OS**: Won't compile; there is no receiver variable
- **Linux OS**: Allowed, but the return value is ignored

The correct form is to declare the receiver variable in a separate statement, as in Example 1.

Example 3: Spawn a function used as an argument to another function.

```
g(cilk_spawn f());
```

- **Windows OS**: Won't compile — There is no receiver variable
- **Linux OS**: Allowed with a warning. `f()` is called, not spawned. There is no parallelism

The correct syntax in this case is to declare the receiver variable in a separate statement, spawn, sync (next section), and use the result as the argument to g(). However, there is no benefit to this as the parent strand must sync immediately and cannot run in parallel with f().

```
int x;
x = cilk_spawn f();
cilk_sync;
g(x);
```

CILK_SPAWN RESTRICTIONS (WINDOWS* OS ONLY)

The Intel Cilk++ compiler for Windows OS requires a receiver for all non-void spawned functions.

Taken extra care with pass-by-const-reference arguments which are bound to rvalues. Rvalues are temporary variables that hold the intermediate results of an expression (e.g., the return value of a function call within a more complex expression). Rvalue temporaries are destroyed at the end of the full expression. An rvalue in the argument list of a spawned function is likely to be destroyed before the function returns, yielding a race condition. This limitation may be relaxed in the future. To avoid this race, eliminate the use of (anonymous) temporaries in a spawned function's argument list by storing the temporary values into named variables. For example, convert this:

```
extern std::string f(int);
extern int doit(const std::string& s);
x = cilk_spawn doit(f(y));
```

to this:

```
extern std::string f(int);
extern int doit(const std::string& s);
std::string a = f(y);
x = cilk_spawn doit(a);
```

The `cilk_sync` corresponding to this `cilk_spawn` must occur before the temporary variable (a in this case) goes out of scope.

CILK_SYNC

The `cilk_sync` statement indicates that the current function cannot run in parallel with its spawned children. After the children all complete, the current function can continue.

The syntax is as follows:

```
cilk_sync;
```

`cilk_sync` only syncs with children spawned by this function. Children of other functions are not affected.

There is an implicit `cilk_sync` at the end of every function and every try block that contains a `cilk_spawn`. The Cilk++ language is defined this way for several reasons:

- To ensure that program resource use does not grow out of proportion to the program's parallelism.
- To ensure that a race-free parallel program has the same behavior as the corresponding serial program. An ordinary non-spawn call to a function works the same regardless of whether the called function spawns internally.
- There will be no strands left running that might have side effects or fail to free resources.
- The called function will have completed all operations when it returns.

See ***The Power of Well-Structured Parallelism (answering a FAQ about Cilk++)*** (http://software.intel.com/en-us/articles/The-Power-of-Well-Structured-Parallelism-answering-a-FAQ-about-Cilk) for more discussion of this point.

CILK_FOR

A `cilk_for` loop is a replacement for the normal C++ `for` loop that permits loop iterations to run in parallel. The Cilk++ compiler converts a `cilk_for` loop into an efficient divide-and-conquer recursive traversal over the loop iterations.

Sample `cilk_for` loops are:

```
cilk_for (int i = begin; i < end; i += 2)
    f(i);

cilk_for (T::iterator i(vec.begin()); i != vec.end(); ++i)
    g(i);
```

The "***serialization*** (Page 126)" of a valid Cilk++ program has the same behavior as the similar C++ program, where the serialization of `cilk_for` is the result of replacing "`cilk_for`" with "`for`". Therefore, a `cilk_for` loop must be a valid C++ `for` loop, but `cilk_for` loops have several constraints compared to C++ `for` loops.

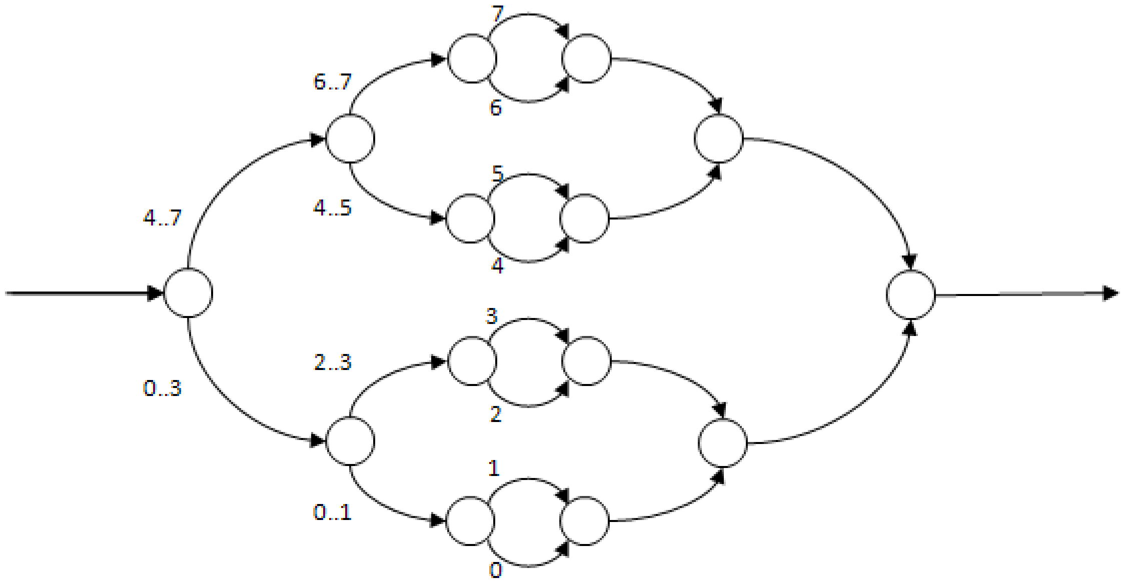
Since the loop body is executed in parallel, it must not modify the control variable nor should it modify a ***nonlocal variable*** (Page 124), as that would cause a ***data race*** (Page 123). The cilkscreen race detector will help detect these data races.

Serial/parallel structure of cilk_for

Note that using `cilk_for` is *not* the same as spawning each loop iteration. In fact, the Cilk++ compiler converts the loop body to a function that is called recursively using a divide-and-conquer strategy allows the Cilk++ scheduler to provide significantly better performance. The difference can be seen clearly in the DAG for the two strategies.

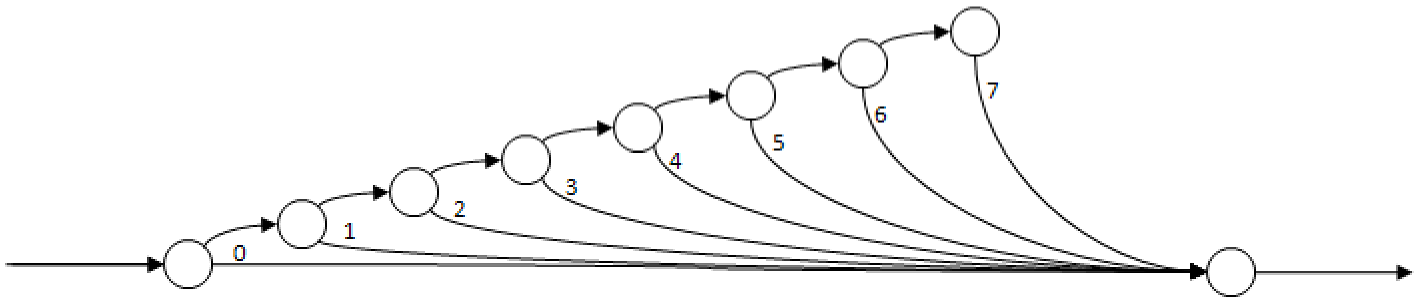
First, the DAG for a cilk_for, assuming N=8 iterations and a grain size of 1. The numbers labeling the strands indicate which loop iteration is handled by each strand. Note that at each division of work, half of the remaining work is done in the child and half in the continuation. Importantly, the overhead of both the loop itself and of spawning new work is divided evenly along with the cost of the loop body.

If each iteration takes the same amount of time T to execute, then the span is $\log_2(N)$ * T, or 3 * T for 8 iterations. The run-time behavior is well balanced regardless of the number of iterations or number of workers.



Serial/parallel structure when spawning within a serial loop

Here is the DAG for a serial loop that spawns each iteration. In this case, the work is not well balanced, because each child does the work of only one iteration before incurring the scheduling overhead inherent in entering a sync. For a short loop, or a loop in which the work in the body is much greater than the control and spawn overhead, there will be little measurable performance difference. However, for a loop of many cheap iterations, the overhead cost will overwhelm any advantage provided by parallelism.



CILK_FOR SYNTAX

The general `cilk_for` syntax is:

```
   cilk_for (declaration;
             conditional expression;
             increment expression)
       body
```

- The *declaration* must declare and initialize a single variable, called the "control variable". The constructor's syntactic form does not matter. If the variable type has a default constructor, no explicit initial value is needed.
- The *conditional expression* must compare the control variable to a "termination expression" using one of the following comparison operators:

  ```
      <   <=   !=   >=   >
  ```

  The termination expression and control variable can appear on either side of the comparison operator, but the control variable cannot occur in the termination expression. The termination expression value must not change from one iteration to the next.
- The *increment expression* must add to or subtract from the control variable using one of the following supported operations:

  +=

  -=

  ++ (prefix or postfix)

  -- (prefix or postfix)

  The value added to (or subtracted from) the control variable, like the loop termination expression, must not change from one iteration to the next.

CILK_FOR TYPE REQUIREMENTS

With care, you may use custom data types for the `cilk_for` control variable. For each custom data types, you need to provide some methods to help the runtime system compute the loop range size so that it can be divided. Types such as integer types and STL random-access iterators have an integral difference type already, and so require no additional work.

Suppose the control variable is declared with type `variable_type` and the loop termination expression has type `termination_type`; for example:

```
extern termination_type end;
extern int incr;
cilk_for (variable_type var; var != end; var += incr) ;
```

You must provide one or two functions to tell the compiler how many times the loop executes; these functions allow the compiler to compute the integer difference between `variable_type` and `termination_type` variables:

```
difference_type operator-(termination_type, variable_type);
difference_type operator-(variable_type, termination_type);
```

- The argument types need not be exact, but must be convertible from `termination_type` or `variable_type`.
- The first form of `operator-` is required if the loop could count up; the second is required if the loop could count down.
- The arguments may be passed by `const` reference or value.
- The program will call one or the other function at runtime depending on whether the increment is positive or negative.
- You can pick any integral type as the `difference_type` return value, but it must be the same for both functions.

- It does not matter if the `difference_type` is signed or unsigned.

Also, tell the system how to add to the control variable by defining:

```
variable_type operator+(variable_type, difference_type);
```

If you wrote "-=" or "`--`" instead of "+=" or "++" in the loop, define `operator-` instead.

Finally, these operator functions must be consistent with ordinary arithmetic. The compiler assumes that adding one twice is the same as adding two once, and if

```
X - Y == 10
```

then

```
Y + 10 == X
```

CILK_FOR RESTRICTIONS

In order to parallelize a loop using the "divide-and-conquer" technique, the runtime system must pre-compute the total number of iterations and must be able to pre-compute the value of the loop control variable at every iteration. To enable this computation, the control variable must act as an integer with respect to addition, subtraction, and comparison, even if it is a user-defined type. Integers, pointers, and random access iterators from the standard template library all have integer behavior and thus satisfy this requirement.

In addition, a `cilk_for` loop has the following limitations, which are not present for a standard C++ `for` loop. The compiler will report an error or warning for most of these errors.

- There must be exactly one loop control variable, and the loop initialization clause must assign the value. The following form is *not* supported:
  ```
  cilk_for (unsigned int i, j = 42; j < 1; i++, j++)
  ```
- The loop control variable must not be modified in the loop body. The following form is *not* supported:
  ```
  cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
  ```
- The termination and increment values are evaluated once before starting the loop and will not be re-evaluated at each iteration. Thus, modifying either value within the loop body will not add or remove iterations. The following form is *not* supported:
  ```
  cilk_for (unsigned int i = 1; i < x; ++i) x = f();
  ```
- The control variable must be declared in the loop header, not outside the loop. The following form is *not* supported:
  ```
  int i; cilk_for (i = 0; i < 100; i++)
  ```
- A `break` or `return` statement will *NOT* work within the body of a `cilk_for` loop; the compiler will generate an error message. `break` and `return` in this context are reserved for future speculative parallelism support.
- A `goto` can only be used within the body of a `cilk_for` loop if the target is within the loop body. The compiler will generate an error message if there is a `goto` transfer into or out of a `cilk_for` loop body. Similarly, a `goto` cannot jump into the body of a `cilk_for` loop from outside the loop.
- A `cilk_for` loop may not be used in a constructor or destructor. It may be used in a function called from a constructor or destructor.

- A cilk_for loops may not "wrap around". For example, in C++ you can write:
- for (unsigned int i = 0; i != 1; i += 3);
  and this has well-defined, if surprising, behavior; it means execute the loop 2,863,311,531 times. Such a loop produces unpredictable results in Cilk++ when converted to a cilk_for.
- A cilk_for may not be an infinite loop such as:
  `cilk_for (unsigned int 1 = 0; i != i; i += 0);`

CILK_FOR GRAIN SIZE

The cilk_for statement divides the loop into chunks containing one or more loop iterations. Each chunk is executed serially, and is spawned as a chunk during the execution of the loop. The maximum number of iterations in each chunk is the *grain size*.

In a loop with many iterations, a relatively large grain size can significantly reduce overhead. Alternately, with a loop that has few iterations, a small grain size can increase the parallelism of the program and thus improve performance as the number of processors increases.

Setting the Grain Size

Use the cilk_grainsize pragma to specify the grain size for one cilk_for loop:

```
#pragma cilk_grainsize = expression
```

For example, you might write:

```
#pragma cilk_grainsize = 1
cilk_for (int i=0; i<IMAX; ++i) { . . . }
```

If you do not specify a grain size, the system calculates a default that works well for most loops. The default value is set as if the following pragma were in effect:

```
#pragma cilk_grainsize = min(512, N / (8*p))
```

where N is the number of loop iterations, and p is the number of workers created during the current program run. Note that this formula will generate parallelism of at least 8 and at most 512. For loops with few iterations (less than 8 * workers) the grain size will be set to 1, and each loop iteration may run in parallel. For loops with more than (4096 * p) iterations, the grain size will be set to 512.

If you specify a grain size of zero, the default formula will be used. The result is undefined if you specify a grain size less than zero.

Note that the expression in the pragma is evaluated at run time. For example, here is an example that sets the grain size based on the number of workers:

```
#pragma cilk_grainsize = n/(4*cilk::current_worker_count())
```

Loop Partitioning at Run Time

The number of chunks that are executed is approximately the number of iterations N divided by the grain size K.

The Cilk++ compiler generates a divide-and-conquer recursion to execute the loop. In pseudo-code, the control structure looks like this:

```
void run_loop(first, last)
{
    if (last - first) < grainsize)
    {
        for (int i=first; i<last ++i) LOOP_BODY;
    }
    else
    {
        int mid = (last-first)/2;
        cilk_spawn run_loop(first, mid);
        run_loop(mid, last);
    }
}
```

In other words, the loop is split in half repeatedly until the chunk remaining is less than or equal to the grain size. The actual number of iterations run as a chunk will often be less than the grain size.

For example, consider a cilk_for loop of 16 iterations:

```
    cilk_for (int i=0; i<16; ++i) { ... }
```

With grain size of 4, this will execute exactly 4 chunks of 4 iterations each. However, if the grain size is set to 5, the division will result in 4 unequal chunks consisting of 5, 3, 5 and 3 iterations.

If you work through the algorithm in detail, you will see that for the same loop of 16 iterations, a grain size of 2 and 3 will both result in exactly the same partitioning of 8 chunks of 2 iterations each.

Selecting a Good Grain Size Value

The default grain size usually performs well. However, here are guidelines for selecting a different value:

- If the amount of work per iteration varies widely and if the longer iterations are likely to be unevenly distributed, it might make sense to reduce the grain size. This will decrease the likelihood that there is a time-consuming chunk that continues after other chunks have completed, which would result in idle workers with no work to steal.
- If the amount of work per iteration is uniformly small, then it might make sense to increase the grain size. However, the default usually works well in these cases, and you don't want to risk reducing parallelism.
- If you change the grain size, carry out performance testing to ensure that you've made the loop faster, not slower.
- Use `cilkview` to estimate a program's ***work*** (Page 127), ***span*** (Page 126), and spawn overhead. This information can help determine the best granularity and whether it is appropriate to override the default grain size.

Several ***examples*** (Page 28) use the grain size pragma:

- `matrix-transpose`
- `cilk-for`
- `sum-cilk`

CILK++ AND C++ LANGUAGE LINKAGE

A function using the Cilk++ calling convention is said to have *Cilk++ language linkage* and is known as a *Cilk++ function*. A function using a C++ or C calling convention is said to have *C++* or *C language linkage* and is known as a *C++* or *C function.*

A Cilk++ function can use Cilk++ keywords and can call C++ or C functions directly. The reverse, however, is not true. A C or C++ function cannot call a Cilk++ function directly, nor can it use Cilk++ keywords.

A later chapter, "***Mixing C++ and Cilk++*** (Page 81)", shows how to call Cilk++ functions from C++ code. This is often useful in large applications.

Windows* Only: Please note that pointers to members of classes with virtual base classes (data or functions) cannot be used as arguments to Cilk++ functions.

DECLARATIVE REGIONS AND LANGUAGE LINKAGE

The `extern` keyword specifies the linkage in a *declarative region*. The syntax is:

```
extern string-literal { declaration-list }
extern string-literal declaration
```

`string-literal` can be any of "`Cilk++`", "`C++`" or "`C`" to specify the language linkage for the declarations. The string is case-sensitive.

There are several special cases and exceptions:

- The `main()` function always has C language linkage whether or not it is declared using `extern "C"`.
- The `cilk_main()` function always has Cilk++ linkage, whether or not it is declared using `extern "Cilk++"`.
- A program should not contain both `main()` and `cilk_main()`.
  - **Windows* OS:** This will produce a compiler or linker error.
  - **Linux* OS:** If a program contains both `main()` and `cilk_main()`, then `main() will be called`.
- The topmost (default) declarative region of a Cilk++ file, outside of any `extern "Cilk++"/"C++"/"C"` construct, is a Cilk++ declarative region.
- The `__cilk` macro is provided to declare a Cilk++ member function in a C++ class, where `extern Cilk++` would not be valid:
  - `__cilk` is position-sensitive so that `void __cilk foo()` is valid but `__cilk void foo()` is not.
  - The "***Mixing C++ and Cilk++*** (Page 81)" chapter uses `__cilk` in code examples.

Language linkage applies to functions, function types, `struct` types, `class` types, and `union` types. The following rules apply:

- Fundamental and enumeration types do not have language linkage.

- A `typedef` is simply an alias for a type which may or may not have language linkage; the `typedef` itself does not have language linkage.
- There are special rules for templates, as described below in the next section.
- A function with one language linkage has a different type than a function with a different language linkage, even when they have the same prototype.
- Do not cast a function pointer from Cilk++ to C++ or C language linkage or vice versa; the results will be unpredictable and not useful.
- A virtual function overridden in a derived class is required to have the same language linkage in the base and derived classes (including compiler-generated virtual destructors). The Cilk++ Linux compiler will change the linkage of the derived class function to match the base class. The Windows compiler will report an error.
- **Windows OS:** The language linkage for a function can be overridden by declaring it with a `__cilk`, `__cdecl`, `__thiscall`, `__stdcall`, or `__fastcall` calling-convention specifier immediately before the function's name. Except for `__cilk` (which is also valid in the Cilk++ compiler for Linux OS), all these Microsoft-specific calling conventions give a function C++ language linkage.
- **Windows OS:** A class's language linkage applies to member functions that are generated automatically by the compiler when not declared; i.e., the default constructor, copy constructor, assignment operator, and destructor.
- **Linux OS:** A compiler-generated member function has Cilk++ language linkage if and only if it calls a Cilk++ function.
- Conflicting `extern` statements will generate warnings, and the first specification is used. For example, in the following, T1 has C++ linkage:

```
    extern "C++" class T1;
    extern "Cilk++" class T1 { ... };
```

CALLING C++ FUNCTIONS FROM CILK++

Cilk++ functions can call C++ functions. This makes it easier to parallelize existing C++ code.

You must inform the Cilk++ compiler that a specific function is a C++ function (i.e., it has C++ *linkage*) and not a Cilk++ function.

To declare a specific function to have C++ linkage, prefix the function declaration with `extern "C++"`:

```
extern "C++" void *operator new(std::size_t, void* ptr);
```

Multiple declarations can be declared as having C++ linkage by creating a C++ declarative region:

```
extern "C++" {
    void myCppFunction(void*);
    int anotherCppFunction(int);
}
```

Do not, however, have a `#include` statement in an `extern "C++"` block. Doing so will cause a compiler "conflicting specification" error when building the program's ***serialization*** (Page 126).

Including a C++ header in a Cilk++ program requires two Cilk++ macros (defined in <`cilk.h`>):

- `CILK_BEGIN_CPLUSPLUS_HEADERS`
- `CILK_END_CPLUSPLUS_HEADERS`

The macros are null when building the serialization and are otherwise defined to be `extern "C++" {` and `}`, respectively. See the "***Nested*** `#include` ***Statements*** (Page 83)" section for more information.

The correct way to include a C++ header into a Cilk++ program is:

```
CILK_BEGIN_CPLUSPLUS_HEADERS
    #include <mycppheader.h>
CILK_END_CPLUSPLUS_HEADERS
```

The Cilk++ compiler treats system header files as if they were included within an `extern "C++"` region, thus causing declarations within system header files to be given C++ linkage.

Windows* OS: If a file named `.sys_include` is present in an include directory, all header files in that directory are treated as system header files. The installer creates `.sys_include` files in known system include directories.

Linux* OS: If a file is in an include directory specified with the `-isystem` option, `cilk++` will treat that header file as a system header file.

LANGUAGE LINKAGE RULES FOR TEMPLATES

If a function template or member function of a class template is declared within a Cilk++ declarative region, then any instantiation of that function will have Cilk++ language linkage. However, if a function template or member function of a class template is declared within a C++ declarative region, then the special rules listed below apply when instantiating the template. The intent of these rules is to allow C++ templates (such as an STL template) to be instantiated from within Cilk++ code.

- A ***C++ template*** is defined as a class or function template declared within C++ declarative region. A ***Cilk++ template*** is defined as a class or function template declared within a Cilk++ declarative region, including the top-level (default) declarative region.
- A ***Cilk++ type*** is one of the following:
  - A `struct`, `class`, or `union` that is declared within a Cilk++ declarative region
  - An instantiation of a Cilk++ class template
  - A pointer or reference to a Cilk++ type or to a Cilk++ function
  - An array of Cilk++ types
  - A nested class of a Cilk++ type
- An instantiation of a C++ class template is ***promoted*** to a Cilk++ type if any of the template type arguments is a Cilk++ type. Nested classes and member functions of such a promoted type are likewise promoted to Cilk++ language linkage.

- It is possible that the instantiation of a nested class or function template will be promoted to Cilk++ linkage even if its enclosing class is not a Cilk++ type or a promoted C++ class template. However, if a class template instantiation is promoted, all of its nested class and function templates are also promoted.

CILK++ AND THE C++ BOOST LIBRARIES

Using the ***Boost* C++ Libraries*** (http://www.boost.org/) in a Cilk++ program requires care. The Cilk++ language defines rules governing expansion of template functions as C++ or Cilk++. These rules work well for the C++ standard template library. They do not work well with the Boost template library. Cilk++ programs using Boost may cause compiler errors related to calls from C++ into Cilk++.

Compiling Boost templates as Cilk++ instead of C++ will avoid many problems. This means:

- On Linux* OS, install Boost outside of `/usr/include` and other standard include directories which are assumed to contain C and C++ code.
- On Windows* OS, do not create a `.sys_include` file in the Boost include directory.

PREPROCESSOR MACROS

`__cilkplusplus`

This macro is defined automatically by the Cilk++ compiler and is set to the Cilk++ language version number. The value in this release is 100, indicating language version 1.0.

`__cilkartsrev`

This macro is defined automatically by the Cilk++ compiler and is set to the Cilk++ compiler version number. The value in this release is 10100, indicating compiler version 1.1.0.

`__cilkartsbuild`

Compilers provided by Cilk Arts define this macro to the unique internal build number. You should generally not need to use this macro.

`__cplusplus`

Although Cilk++ is a different language than C++, many Cilk++ programs benefit from having this macro from C++ defined. For this reason, the Cilk++ compiler defines `__cplusplus` in addition to `__cilkplusplus`. To detect compilation with a C++ compiler which is *not* a Cilk++ compiler, compose a conditional directive as follows:

```
#if defined(__cplusplus) && ! defined(__cilkplusplus)
```

EXCEPTION HANDLING

The Cilk++ language attempts to reproduce, as closely as possible, the semantics of C++ exception handling. This generally requires limiting parallelism while exceptions are pending, and programs should not depend on parallelism during exception handling.

There is an implicit `cilk_sync` at the end of every `try` block. A function has no active children when it begins execution of a `catch` block.

`cilk_spawn` behavior during exception handling is system-dependent.

Windows* OS: `cilk_spawn` has no effect while an exception is pending, from construction of the exception object to its destruction after the last exception handler finishes.

Linux* OS: `cilk_spawn` has no effect during execution of the exception object destructor, and `cilk_sync` statements are inserted before throws. These restrictions only affect execution of functions that throw and catch exceptions and the functions in between. If function **f** spawns function **g**, function **g** spawns function **h**, and function **h** throws an exception caught by function **g**, then function **f** is not affected by the exception.

Exception logic is:

- If an exception is thrown and not caught in a spawned child, then that exception is rethrown in the parent at the next sync point.
- If the parent or another child also throws an exception, then the first exception that would have been thrown in the serial execution order takes precedence. The logically-later exceptions are discarded. There is currently no mechanism for collecting multiple exceptions thrown in parallel.

Note that throwing an exception does not abort existing children or siblings of the strand in which the exception is thrown; these strands will run normally to completion. This behavior may change in a future version of the Intel® Cilk++ SDK.

Windows OS: The Intel Cilk++ compiler support for Microsoft Visual Studio currently supports only C++ (synchronous) exceptions. Attempting to use asynchronous exceptions by using the `/EHa` compiler option will be rejected by the compiler.The compiler does not support the `__try`, `__except`, `__finally`, or `__leave` Microsoft* C++ extensions.

Chapter 7

Cilk++ Execution Model

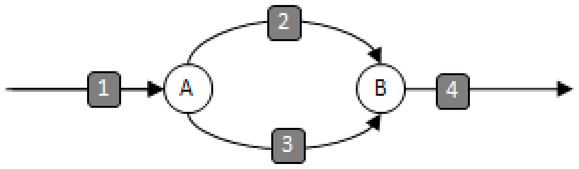
Earlier, we described how a DAG can be used to illustrate the serial/parallel structure of a Cilk+++ program. Recall that the DAG does not depend on the number of processors. The execution model describes how the runtime scheduler maps strands to workers.

When parallelism is introduced, multiple strands may execute in parallel. However, in a Cilk++ program, strands that *may* execute in parallel are not *required* to execute in parallel. The scheduler makes this decision dynamically. We will not explain exactly how the scheduler operates, but the interested reader may consult previously published literature that describes the work-stealing scheduler in great detail.

Consider the following Cilk++ program fragment:

```
do_init_stuff();          // execute strand 1
cilk_spawn func3();            // spawn strand 3   (the "child")
do_more_stuff();          // execute strand 2 (the "continuation")
cilk_sync;
do_final_stuff;           // execute strand 4
```

Here is the simple DAG for the code:



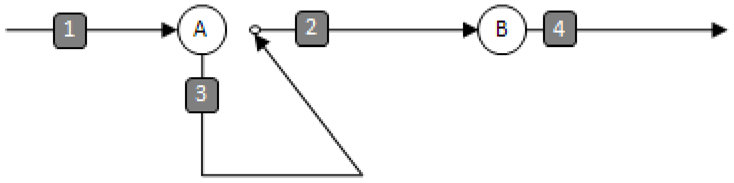
Recall that a ***worker*** (Page 127) is an operating system thread that executes a Cilk++ program. If there is more than one worker available, there are two ways that this program may execute:

- The entire program may execute on a single worker, or
- The scheduler may choose to execute strands (2) and (3) on different workers.

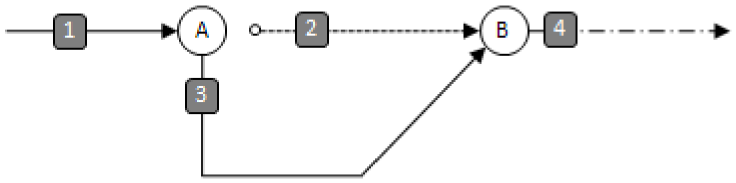
In order to guarantee serial semantics, the function that is spawned (the "child", or strand (3) in this example) is always executed on the same worker that is executing the strand that enters the spawn. Thus, in this case, strand (1) and strand (3) are guaranteed to run on the same worker.

If there is a worker available, then strand (2) (the "continuation") may execute on a different worker. We call this a *steal*, and say that the continuation was *stolen* by the new worker.

To illustrate these two execution options, we introduce a new diagram. First, we illustrate the execution on a single worker:



If a second worker is scheduled, the second worker will begin executing the continuation, strand (2). The first worker will proceed to the sync at (B). Here, we indicate the second worker by illustrating strand (2) with a dotted line. After the sync, strand (4) may continue on either worker. In the current implementation, strand (4) will execute on the last worker that reaches the sync.



The details of the execution model have several implications that will be described later, when we discuss the interaction between Cilk++ workers and system threads, and when we describe reducers. For now, the key ideas to remember are:

- After a cilk_spawn, the child will always execute on the same worker (i.e. system thread) as the caller.
- After a cilk_spawn, the continuation may execute on a different worker. If this occurs, we say that the continuation was *stolen* by another worker.
- After a cilk_sync, execution may proceed on any worker that executed a strand that entered the sync.

Chapter 8

Reducers

This chapter describes Cilk++ reducers, their use, and how to develop custom reducers.

The Intel® Cilk++ SDK provides ***reducers*** to address the problem of accessing nonlocal variables in parallel code. See "Reducers: Introduction to Avoiding Races" for a simple example, and these articles: ***"Are Determinacy-Race Bugs Lurking in YOUR Multicore Application?"*** (http://software.intel.com/en-us/articles/Are-Determinacy-Race-Bugs-Lurking-in-YOUR-Multicore-Application) and ***"Global Variable Reconsidered"*** (http://software.intel.com/en-us/articles/Global-Variable-Reconsidered) provide additional examples. Note that these papers describe an older reducer syntax, but the concepts are still applicable.

Conceptually, a reducer is a variable that can be safely used by multiple strands running in parallel. The runtime system ensures that each worker has access to a private copy of the variable, eliminating the possibility of races without requiring locks. When the strands synchronize, the reducer copies are merged (or "reduced") into a single variable. The runtime system creates copies only when needed, minimizing overhead.

Reducers have several attractive properties:

- Reducers allow reliable access to nonlocal variables without races.
- Reducers do not require locks and therefore avoid the problem of lock contention (and subsequent loss of parallelism) that arises from using locks to protect nonlocal variables.
- Defined and used correctly, reducers *retain serial semantics*. The result of a Cilk++ program that uses reducers is the same as the serial version, and the result does not depend on the number of processors or how the workers are scheduled. Reducers can be used without significantly restructuring existing code.
- Reducers are implemented efficiently, incurring minimal overhead.
- Reducers can be used independently of the program's control structure, unlike constructs that are defined over specific control structures such as loops.

Reducers are defined by writing C++ templates that provide an interface to the runtime system.

Reducers are a kind of *hyperobject*. In Cilk Arts Cilk++ version 1.0.3 and earlier, reducers were declared using a hyperobject template. Beginning in Cilk Arts Cilk++ version 1.1.0, reducer declaration has been simplified.

In this chapter, we will:

- Demonstrate how to use a reducer supplied in the Intel Cilk++ SDK.
- Describe the library of reducers.
- Explain how to develop your own reducers.
- Discuss some technical details, including appropriate operations, performance considerations and limitations of reducers.

WARNING: The syntax used to declare and use reducers changed between Cilk++ version 1.0.3 and Cilk++ version 1.1.0. Reducers written in the older style are still supported, but may be removed in a future version of the Intel Cilk++ SDK.

USING REDUCERS — A SIMPLE EXAMPLE

A common need for reducers appears when trying to accumulate a sum in parallel. Consider the following serial program that repeatedly calls a `compute()` function and accumulates the answers into the `total` variable.

```
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i;           // return a value computed from i
}

int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    unsigned int total = 0;

    // Compute the sum of integers 1..n
    for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }

    // the sum of the first n integers should be n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;

    if (total == correct)
            std::cout << "Total (" << total
                      << ") is correct" << std::endl;
    else
            std::cout << "Total (" << total
                      << ") is WRONG, should be "
                      << correct << std::endl;
    return 0;
}
```

Converting this to a Cilk++ program and changing the `for` to a `cilk_for` causes the loop to run in parallel, but creates a ***data race*** (Page 87) on the `total` variable. To resolve the race, we simply make `total` a reducer — in this case, a reducer_opadd, defined for types that have an associative "+" operator. The changes are marked below. This program is provided as `examples/reducer/reducer.cilk`.

```
#include <cilk.h>
#include <reducer_opadd.h>
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i;           // return a value computed from i
}

int cilk_main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    cilk::reducer_opadd<unsigned int> total;

    // Compute 1..n
    cilk_for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }

    // the sum of the first N integers should be n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;

    if (total.get_value() == correct)
            std::cout << "Total (" << total.get_value()
                      << ") is correct" << std::endl;
    else
            std::cout << "Total (" << total.get_value()
                      << ") is WRONG, should be "
                      << correct << std::endl;
    return 0;
}
```

The changes in the serial code show how to use a reducer provided by the Intel® Cilk++ SDK:

- Include the appropriate reducer header file.
- Declare the reduction variable as a `reducer_kind<TYPE>` rather than a TYPE.
- Introduce parallelism, in this case by changing the `for` loop to a `cilk_for` loop.
- Retrieve the reducer's terminal value with the `get_value()` method after the `cilk_for` loop is complete.

Two ***examples*** (Page 28), `reducer` and `sum-cilk` use `reducer_opadd` in the same manner as in this code fragment.

Noet: Reducers are objects. As a result, they cannot be copied directly. The results are unpredictable if you copy a reducer object using `memcpy()`. Instead, use a copy constructor.

HOW REDUCERS WORK

In this section, we discuss in more detail the mechanisms and semantics of reducers. This information should help the more advanced programmer understand more precisely what rules govern the use of reducers as well as provide the background needed to write custom reducers.

In the simplest form, a reducer is an object that has a value, an identity, and a reduction function.

The reducers provided in the reducer library provide additional interfaces to help ensure that the reducers are used in a safe and consistent fashion.

In this discussion, we refer to the object created when the reducer is declared as the "leftmost" instance of the reducer.

In the following sections, we present a simple example and discuss the run-time behavior of the system as this program runs.

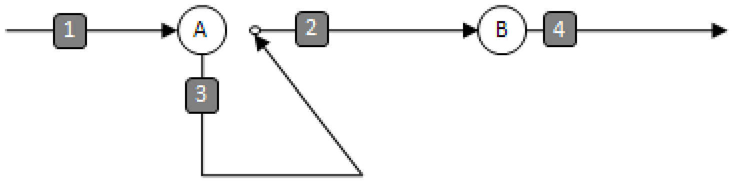
First, consider the two possible executions of a `cilk_spawn`, with and without a steal. The behavior of a reducer is very simple:

- If no steal occurs, the reducer behaves like a normal variable.
- If a steal occurs, the continuation receives a view with an identity value, and the child receives the reducer as it was prior to the spawn. At the corresponding sync, the value in the continuation is merged into the reducer held by the child using the reduce operation, the new view is destroyed, and the original (updated) object survives.

The following diagrams illustrate this behavior:

No steal

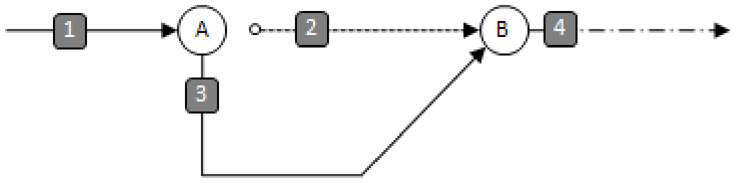
If there is no steal after the `cilk_spawn` indicated by (A):



In this case, a reducer object visible in strand (1) can be directly updated by strand (3) and (4). There is no steal, thus no new view is created and no `reduce` operation is called.

Steal

If strand (2), the continuation of the `cilk_spawn` at (A), is stolen:



In this case, a reducer object in strand (1) is visible in strand (3), the child. Strand (2), the continuation, receives a new view with an identity value. At the sync (B), the new reducer view is reduced into the original view visible to strand (3).

Example: Using reducer_opadd<>

Here is a simplified program that uses reducer_opadd<> to accumulate a sum of integers in parallel. For addition, the identity value is -, and the reduction function adds the right value into the left value:

```
1   reducer_opadd<int> sum;
2
3   void addsum()
4   {
5       sum += 1;
6   }
7
8   int cilk_main()
9   {
10      sum += 1;
11      cilk_spawn addsum();
12      sum += 1;
        // the value of sum here depends on whether a steal occured
13      cilk_sync;
14      return sum.get_value();
15  }
```

If no steal occurs...

First consider the serial case when the execution occurs on a single processor, and there is no steal. In this case, there is no private view of `sum` created, so all operations are performed on the leftmost instance. Because no new views are created, the reduction operation is never called. The value of `sum` will increase monotonically from 0 to its final value of 3.

In this case, because there was no steal, the `cilk_sync` statement is treated as a no-op.

If a steal occurs...

If a steal occurs, then, when `sum` is accessed at line 12, a new view with an identity value (0) is created. In this case, the value of sum after line 12 executes will be 1. Note that the parent gets the new (identity) view and child gets the view that was active at the time of the spawn. This allows reducers to maintain deterministic results for reduce operations that are associative but not cummutative. The child (addsum) operates on the leftmost instance, and so sum increases from 1 to 2 at line 5.

When the `cilk_sync` statement is encountered, if the strands joining together have different views of sum, those views will be merged using the reduction operation. In this case, reduction is an addition, so the new view in the parent (value 1) is added into the view held by the child (value 2) resulting in the leftmost instance receiving the value 3. After the reduction, the new view is destroyed.

Lazy semantics

It is conceptually correct to understand that each strand has a private view of the reducer. For performance purposes, these views are created lazily—that is, only when two conditions are met.

- First, a new view will only be created after a steal.
- Second, the new view is created when the reducer is first accessed in the new strand. At that point, a new instance is created, holding an identify value as defined by the default constructor for the type.

If a new view has been created, it is merged into the prior view at `cilk_sync`. If no view was created, no reduction is necessary. (Logically, you can consider that an identity was created and then merged, which would be a no-op.)

Safe operations

It is possible to define a reducer by implementing only the identity and reduction functions. However, it is typically both safer and more convenient to provide functions using operator overloads in order to restrict the operations on reducers to those that make sense.

For example, `reducer_opadd` defines +=, -=, * ++, --, +, and - operators. Operations such as multiply (*) and divide (/) will not provide deterministic and consistent semantics, and are thus not provided in the `reducer_opadd` definition.

SAFETY AND PERFORMANCE CAUTIONS

In general, reducers provide a powerful way to define global variables that do not require locks and that provide results across parallel runs that are repeatable and exactly the same as the results of a serial execution.

However, there are some cautions to be aware of.

Safety

To get strictly deterministic results, all operations (update and merge) that modify the value of a reducer must be associative.

The reducers defined in the reducer library provide operators that are associative. In general, if you only use these operators to access the reducer, you will get deterministic, serial semantics. It is possible to use reducers with operations that are not associative, either by writing your own reducer with non-associative operations, or by accessing and updating the underlying value of any reducer with unsafe operations.

Determinism

When reducers are instantiated with floating-point types, the operations are not strictly associative. Specifically, the order of operations can generate different results when the exponents of the values differ. This can lead to results that vary based on the order in which the strands execute. For some programs, these differences are tolerable, but be aware that you may not see exactly repeatable results between program runs.

Performance

When used judiciously, reducers can incur little or no runtime performance cost. However, the following situations may have significant overhead. Note that the overhead is also proportional to the number of steals that occur.

If you create a large number of reducers (for example, an array or vector of reducers) you must be aware that there is an overhead at steal and reduce that is proportional to the number of reducers in the program.

If you define reducers with a large amount of state, note that it may be expensive to create identity values when the reducers are referenced after a steal.

In addition, if the merge operation is expensive, remember that a merge occurs at every sync that follows a successful steal.

REDUCER LIBRARY

The reducer library in the Intel® Cilk++ SDK contains the reducers shown in the following table.

Each reducer is described in detail in comments in the corresponding header file.

The middle column shows each reducer's identity element and Update operation (there may be several). The next section explains these concepts.

REDUCER/HEADER FILE	IDENTITY/ UPDATE	DESCRIPTION
`reducer_list_append` `<reducer_list.h>`	empty list `push_back()`	Creates a list using an append operation. The final list will always have the same order as the list constructed by the equivalent serial program, regardless of the worker count or the order in which the workers are scheduled.
`reducer_list_prepend` `<reducer_list.h>`	empty list `push_front()`	Creates a list using a prepend operation.
`reducer_max` `<reducer_max.h>`	Argument to constructor `cilk::max_of`	Finds the maximum value over a set of values. The constructor argument has an initial maximum value.
`reducer_max_index` `<reducer_max.h>`	Arguments to constructor `cilk::max_of`	Finds the maximum value and the index of the element containing the maximum value over a set of values. The constructor argument has an initial maximum value and index.
`reducer_min` `<reducer_min.h>`	Argument to constructor `cilk::min_of`	Finds the minimum value over a set of values. The constructor argument has an initial minimum value.

REDUCER/HEADER FILE	IDENTITY/ UPDATE	DESCRIPTION
`reducer_min_index` `<reducer_min.h>`	Arguments to constructor `cilk::min_of`	Finds the minimum value and the index of the element containing the minimum value over a set of values. The constructor argument has an initial minimum value and index.
`reducer_opadd` `<reducer_opadd.h>`	`0` `+=, =, -=, ++, --`	Performs a sum.
`reducer_opand` `<reducer_opand.h>`	`1 / true` &, &=, =	Perform logical or bitwise AND.
`reducer_opor` `<reducer_opor.h>`	`0 / false` |, |=, =	Perform logical or bitwise OR.
`reducer_opxor` `<reducer_opxor.h>`	`0 / false` `^, ^=, =`	Perform logical or bitwise XOR.
`reducer_ostream` `<reducer_ostream.h>`	Arguments to constructor <<	Provides an output stream that can be written in parallel. In order to preserve a consistent order in the output stream, output will be buffered by the reducer class until there is no more pending output to the left of the current position. This ensures that the output will always appear in the same order as the output generated by the equivalent serial program.
`reducer_basic_string` `<reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Creates a string using append or += operations. Internally, the string is maintained as a list of substrings in order to minimize copying and memory fragmentation. The substrings are assembled into a single output string when `get_value()` is called.
`reducer_string` `<reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Provides a shorthand for a `reducer_basic_string` of type `char`.
`reducer_wstring` `<reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Provides a shorthand for a `reducer_basic_string` of type `wchar`.

USING REDUCERS — ADDITIONAL EXAMPLES

The following sections illustrate how to use a variety of reducers, including the String and List reducers included with the Intel® Cilk++ SDK.

STRING REDUCER

`reducer_string` builds 8-bit character strings, and the example uses += (string concatenation) as the update operation.

This example demonstrates how reducers work with the runtime to preserve serial semantics. In a serial for loop, the reducer will concatenate each of the characters 'A' to 'Z', and then print out:

```
The result string is: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The `cilk_for` loop will use a divide-and-conquer algorithm to break this into two halves, and then break each half into two halves, until it gets down to a "reasonable" size chunk of work. Therefore, the first worker might build the string "ABCDEF", the second might build "GHIJKLM", the third might build "NOPQRS", and the fourth might build "TUVWXYZ". The runtime system will always call the reducer's `reduce` method so that the final result is a string containing the letters of the English alphabet in order.

String concatenation is associative (but not commutative), the order of operations is not important. For instance, the following two expressions are equal:

- `"ABCDEF" concat ("GHIJKLM" concat ("NOPQRS" concat "TUVWXYZ"))`
- `("ABCDEF" concat "GHIJKLM") concat ("NOPQRS" concat "TUVWXYZ")`

The result is always the same, regardless of how `cilk_for` creates the work chunks.

The call to `get_value()` performs the reduce operation and concatenates the substrings into a single output string. Why do we use `get_value()` to fetch the string? It makes you think about whether fetching the value at this time makes sense. You *could* fetch the value whenever you want, but, in general, you *should not*. The result might be an unexpected intermediate value, and, in any case, the intermediate value is meaningless. In this example, the result might be "GHIJKLMNOPQRS", the concatenation of "GHIJKLM" and "NOPQRS".

While Cilk++ reducers provide serial semantics, the serial semantics are only guaranteed at the end of the parallel calculation, such as at the end of a `cilk_for` loop, after the runtime system has performed all the reduce operations. *Never* call `get_value()` within the `cilk_for` loop; the value is unpredictable and meaningless since results from other loop iterations are being combined (reduced) with the results in the calling iteration.

Unlike the previous example, which adds integers, the reduce operation is not commutative. You could use similar code to append (or prepend) elements to a list using the reducer library's `reducer_list_append`, as is shown in the example in the next section.

```
#include <reducer_string.h>

int cilk_main()
{
    // ...

    cilk::reducer_string result;
    cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i) {
        result += (char)i;
    }

    std::cout << "The result string is: "
              << result.get_value() << std::endl;

    return 0;
}
```

In this and other examples, each loop iteration only updates the reducer once; however, you could have several updates in each iteration. For example:

```
    cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i) {
        result += (char)i;
        result += tolower((char)i);
    }
```

is valid and would produce the string:

```
    AaBb...Zz
```

LIST REDUCER (WITH USER-DEFINED TYPE)

reducer_list_append creates lists, using the STL list append method as the update operation. The identity is the empty list. The example here is almost identical to the previous string example. The reducer_list_append declaration does, however, require a type, as shown in the following code.

```
    #include <reducer_list.h>

    int cilk_main()
    {
        // ...

        cilk::reducer_list_append<char>  result;
        cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i) {
            result.push_back((char)i);
        }

        std::cout << "String = ";
        std::list<char> r;
        r = result.get_value();
        for (std::list<char>::iterator i = r.begin();
                                       i != r.end(); ++i) {
            std::cout << *i;
        }
        std::cout << std::endl;
    }
```

Reducer with User-Defined Type

Note that the type of the `reducer_list_append` in this example is `char`. The following will generate a compile-time error indicating `invalid override of non-Cilk function with Cilk function.`

```
typedef struct complex_type {
    int x;
    int y;
} complex_type;

cilk::reducer_list_append<complex_type> result;   // ERROR REPORTED
HERE
```

This error is reported because the constructor and destructor of `complex_type`, which are called by the runtime system to create and destroy views, are functions with Cilk++ linkage. To force those functions to have C++ linkage, declare the type in a C++ region as follows:

```
extern "C++" {
  typedef struct complex_type {
      int x;
      int y;
  } complex_type;
}

cilk::reducer_list_append<complex_type> result;
```

For more details, see ***Cilk++ and C++ Language Linkage*** (Page 45).

REDUCERS IN RECURSIVE FUNCTIONS

The previous reducer examples all performed the update operations within a `cilk_for` loop, but reducers work with arbitrary control flow, such as recursively spawned functions. This example illustrates how a reducer can be used to create an in-order list of elements in a tree. Note that the final list will always contain the elements in the same order as in a serial execution, regardless of the number of cores or how the computation is scheduled. There is a similar example in the article: ***Global Variable Reconsidered*** (http://software.intel.com/en-us/articles/Global-Variable-Reconsidered).

```
#include <reducer_list.h>
Node *target;

cilk::reducer_list_append<Node *> output_list;
...
// Output the tree with an in-order walk
void walk (Node *x)
{
    if (NULL == x)
        return;
    cilk_spawn walk (x->left);
    output_list.push_back (x->value);
```

```
        walk (x->right);
    }
```

HOW TO DEVELOP A NEW REDUCER

You can develop a custom reducer if none of the reducers provided in the Intel® Cilk++ SDK satisfy your requirements.

Examples

Any of the reducers in the Intel Cilk++ SDK can be used as models for developing new reducers, although these examples are all relatively complex. The implementations are in the `reducer_*.h` header files in the `include` directory of the installation. Two of the ***Cilk++ examples*** (Page 28) contain simpler custom reducers:

- The "`hanoi`" example program contains a custom reducer that builds the list of moves used to solve the problem using recursive divide-and-conquer, rather than a `cilk_for` loop.
- The "`linear-recurrence`" example has a custom reducer to allow a linear recurrence to be computed in parallel.

Two additional examples are provided in the following sections.

Components of a Reducer

A reducer can be broken into 4 logical parts:

- A "View" class – This is the private data for the reducer. The constructor and destructors must be public so they can be called by the runtime system. The constructor should initialize the View to the "identity value" for your reducer. Identity values will be discussed more below. The View class may make the enclosing reducer class a friend so it can access private data members, or it can provide access methods.
- A "Monoid" class. A monoid is a mathematical concept that will be formally defined below. For now, just accept that your Monoid class must derive from `cilk::monoid_base<View>`, and contain a public static member named reduce with the following signature:
  ```
  static void reduce (View *left, View *right);
  ```
- A hyperobject that provides per-strand views. It should be a private member variable declared as follows:
  private:
  ```
  cilk::reducer<Monoid> imp_;
  ```
- The rest of the reducer, which provides routines to access and modify the data. By convention there is a `get_value()` member which returns the value of the reducer.

Note that the reducers in the reducer library use `struct` instead of `class` for the View and Monoid classes. Recall that the only difference between `struct` and `class` in C++ is that the default access for a `class` is `private`, while the default access for a `struct` is `public`.

The Identity Value

The *identity value* is that value, which when combined with another value *in either order* (that is, a "two-sided identity") produces that second value. For example:

- `0` is the identity value for addition:
  `x = 0 + x = x + 0`
- `1` is the identity value for multiplication:
  `x = 1 * x = x * 1`.
- The empty string is the identity value for string concatenation:
  `"abc" = "" concat "abc" = "abc" concat ""`

The Monoid

In mathematics, a *monoid* comprises a set of values (type), an associative operation on that set, and an identity value for that set and that operation. So for example (integer, +, 0) is a monoid, as is (real, *, 1).

In the Cilk++ language, a Monoid is defined by a type T along with five functions:

`reduce(T *left, T *right)`	evaluates `*left = *left OP *right`
`identity(T *p)`	constructs IDENTITY value into the uninitialized `*p`
`destroy(T *p)`	calls the destructor on the object pointed to by `p`
`allocate(size)`	returns a pointer to `size` bytes of raw memory
`deallocate(p)`	deallocates the raw memory at `p`

These five functions must be either `static` or `const`. A class that meets the requirements of Cilk++ Monoid is usually stateless, but will sometimes contain state used to initialize the identity object.

The `monoid_base` class template is a useful base class for a large set of Monoid classes for which the identity value is a default-constructed value of type T, allocated using operator new. A class derived from `monoid_base` need only declare and implement the `reduce` function.

The `reduce` function merges the data from the "right" instance into the "left" instance. After the runtime system calls the `reduce` function, it will destroy the right instance.

For deterministic results, the `reduce` function must implement an *associative* operation, though it does not need to be *commutative*. If implemented correctly, the `reduce` function will retain serial semantics. That is, the result of running the application serially, or using a single worker, is the same as running the application with multiple workers. The runtime system, together with the associativity of the reduce function ensures that the results will be the same, regardless of the number of workers or processors, or how the strands are scheduled.

WRITING REDUCERS — A "HOLDER" EXAMPLE

This example shows how to write a reducer that is about as simple as possible—it has no update methods, and the `reduce` method does nothing. A "Holder" is analogous the "Thread Local Storage", but without the pitfalls described in the ***OS Thread*** section.

The rule that you should not call `get_value()` except when fully synched is intentionally violated here.

Such a reducer has a practical use. Suppose there is a global temporary buffer used by each iteration of a `for` loop. This is safe in a serial program, but unsafe if the `for` loop is converted to a parallel `cilk_for` loop. Consider the following program that reverses an array of `point` elements, using a global temporary variable `temp` while swapping values.

```
class point
{
public:
    point() : x_(0), y_(0), valid_(false) {};

    void set (int x, int y) {
        x_ = x;
        y_ = y;
        valid_ = true;
    }

    void reset() { valid_ = false; }

    bool is_valid() { return valid_; }
    int x() { if (valid_) return x_; else return -1; }
    int y() { if (valid_) return y_; else return -1; }

private:
    int x_;
    int y_;
    bool valid_;
};

point temp;        // temp is used when swapping two elements

int cilk_main(int argc, char **argv)
{
    int  i;
    point ary[100];

    for (i = 0; i < 100; ++i)
        ary[i].set(i, i);

    cilk_for (int j = 0; j < 100 / 2; ++j)
    {
        // reverse the array by swapping 0 and 99, 1 and 98, etc.
        temp.set(ary[j].x(), ary[j].y());
        ary[j].set (ary[100-j-1].x(), ary[100-j-1].y());
        ary[100-j-1].set (temp.x(), temp.y());
    }

    // print the results
    for (i = 0; i < 100; ++i)
        printf ("%d: (%d, %d)\n", i, ary[i].x(), ary[i].y());

    return 0;
}
```

There is a race on the global variable `temp`, but the serial program will work properly. In this example, it would be simple and safe to declare `temp` inside the `cilk_for()`. However, the "holder" pattern described below can be used to provide a form of storage that is local to Cilk++ strands.

Our solution is to implement and use a "holder" reducer.

First, put the declaration of the `point` class inside an `extern "C++" { }` region. The `point` class is used within the Monoid, so its default constructor and destructor methods must be public and must have C++ linkage, as described in the section discussing how to use a ***reducer with a user-defined type*** (Page 61).

The `point_holder` class is our reducer. It uses the `point` class as the view. The Monoid class contains a reduce which does nothing, because we don't care which version we retain. The rest of the methods of `point_holder` allow us to access and update the reducer data.

```
    extern "C++"
    {
    class point
    {
        // Define the point class here, exactly as above
    };


    // define the point_holder reducer
    class point_holder
    {
        struct Monoid: cilk::monoid_base<point>
        {
            // reduce function does nothing
            static void reduce (point *left, point *right) {}
        };

    private:
        cilk::reducer<Monoid> imp_;

    public:
        point_holder() : imp_() {}
        void set(int x, int y) {
            point &p = imp_.view();
            p.set(x, y);
        }

        bool is_valid() { return imp_.view().is_valid(); }

        int x() { return imp_.view().x(); }

        int y() { return imp_.view().y(); }
    };  // class point_holder

    };  // extern "C++"
```

To use the `point_holder` reducer in the sample program, simply replace the declaration of temp

```
    point temp;             // temp is used when swapping two elements
```

with a declaration using the reducer type:

```
    point_holder temp;  // temp is used when swapping two elements
```

The rest of the original program remains unchanged.

To summarize how the `point_holder` reducer works:

- The default constructor will create a new instance whenever a new `point_holder` is created—that is, whenever `temp` is referenced after a steal.
- The reduce method does nothing. Since we only use `temp` for temporary storage, there is no need to combine (reduce) the left and right instances.
- The default destructor will invoke the destructor, freeing its memory.
- Because the local view value produced in a loop iteration is not combined with values from other iterations, it is valid to call `x() and y()` to get the local values within the `cilk_for`.
- Because the `reduce()` function for the holder does nothing, the default constructor does not need to provide a true identity value.

WRITING REDUCERS — A SUM EXAMPLE

This example shows a slightly more complex custom reducer that includes two update methods.

Different reducers represent different data types and have different update and reducing/merging operations. For example, a list-append reducer would provide a `push_back()` operation, an empty list identity value, and a `reduce` function that performs list concatenation. An integer-max reducer would provide a `max()` operation, a type-specific identity as a constructor argument, and a `reduce` function that keeps the larger of the values being merged.

A reducer can be instantiated on a user-defined class, such as the `Sum` class in the following example or the ***list reducer*** (Page 61) shown earlier. This implementation could be easily generalized to use a template. `Sum` is similar to `reducer_opadd` and is shown to illustrate how to write a custom reducer that supports multiple update operators.

```
extern "C++" {
    class Sum
    {
    public:
        // Required constructor, initialize to identity (0).
        Sum() : d_value() { }
        // Required reduce method
        void reduce(Sum* other) { d_value += other->d_value; }

        // Two update operations
        Sum& operator+=(const int& v) {
            d_value += v; return *this;
        }
        Sum& operator++() {
            ++d_value;
            return *this;
        }

        int get_value() const { return d_value; }

    private:
        int d_value;
    };
}
```

The example illustrates several reducer requirements and features:

- The View class in this reducer is a simple `int`. The default constructor for an `int` initializes it to 0—the identity value for addition.
- The `reduce` function in the monoid simply adds the value of the right instance to the left instance.
- The operations provided are the += and ++ operators.
- We could add variations such as `-=` and `--`, as long as a subsequence of operations starting from the identity can be merged (or "reduced") with another subsequence to produce a value consistent with the entire sequence.
- The `get_value()` function returns the result of the entire sequence of operations. `get_value()` usage is a convention designed to make you think about when you're fetching the reducer's value. While the sum of a sequence of numbers is valid at any time, intermediate values may not be what's expected, nor will they typically be useful.

Chapter 9

Operating System Specific Considerations

This chapter describes:

- How Cilk++ programs interact with operating system threads
- How Cilk++ programs interact with Microsoft Foundation Classes (MFC)
- How to build Linux* Shared Libraries containing Cilk++ code
- How to build Windows* DLLs containing Cilk++ code

USING OTHER TOOLS WITH CILK++ PROGRAMS

Because Cilk++ programs have a stack layout and calling conventions that are different from the standard C++ conventions, tools that understand the binary program executable (including memory checkers such as valgrind, code coverage tools and the like) may not work with the parallel Cilk++ binaries. You can use such programs on the ***serialization*** (Page 25) of the Cilk++ program.

GENERAL INTERACTION WITH OS THREADS

The runtime system allocates a set of OS threads using native OS facilities.

Cilk++ programs do not really always use 100% of all available processors

When running a Cilk++ program, you may observe that the Cilk++ program appears to consume all the resources of all the processors in the system, even when there is no parallel work to perform. This effect is apparent with programs such as the Windows* Task Manager "Performance" tab; all CPUs may appear to be busy, even if only one strand is executing.

In fact, the runtime scheduler does yield the CPUs to other programs. If there are no other programs requesting the processor, then the Cilk++ worker will be immediately run again to look for work to steal, and this is what makes the CPUs appear to be busy. Thus, the CIlk++ program appears to consume all the processors all the time, but there is no adverse effect on the system or other programs.

Use caution when using native threading interfaces

Cilk++ strands are not operating-system threads. A Cilk++ strand will never migrate between workers while running. However, the worker may change after a `cilk_spawn`, `cilk_sync`, or `cilk_for` statement since all these statements terminate one or more strands and create one or more new strands. Furthermore, the programmer does not have any control over which worker will run a specific strand.

This can impact a program in several ways, most importantly:

- Do not use **Windows** thread local storage or **Linux*** Pthreads thread specific data, because the OS thread may change when work is stolen. Instead, use other programming techniques, such as the Cilk++ holder reducer described earlier.

- Do not use operating system locks or mutexes across `cilk_spawn`, `cilk_sync`, or `cilk_for` statements, because only the locking thread can unlock the object. See the "***Holding a Lock Across a Strand Boundary*** (Page 94)" section.

MICROSOFT FOUNDATION CLASSES AND CILK++ PROGRAMS

This section is for ***Windows**** programmers only.

The Microsoft Foundation Classes (MFC) library depends upon thread local storage to map from its class wrappers to the GDI handles for objects. Because a Cilk++ strand is not guaranteed to run on any specific OS thread, Cilk++ code, or code called from a Cilk++ function or method, cannot safely call MFC functions.

There are two methods typically used to perform a computationally-intensive task in an MFC-based application:

- The user interface (UI) thread creates a computation thread to run the computationally-intensive task. The compute thread posts messages to the UI thread to update it, leaving the UI thread free to respond to UI requests.
- The computationally-intensive code is run on the UI thread, updating the UI directly and occasionally running a "message pump" to handle other UI requests.

Since the runtime system can switch operating system threads, Cilk++ code must be isolated from code such as MFC that depends on Thread Local Storage.

To add a computation thread to an MFC program:

- Create a computation thread using operating-system facilities (i.e., `_beginthreadex` or `AfxBeginThread`). All the C++ code that is to be converted to Cilk++ should run in this thread. The computation thread leaves the main (UI) thread available to run the message pump for processing window messages and updating the UI.
- Pass the handle (`HWND`) for the UI windows to the computation thread. When the computation thread needs to update the UI, it shuld send a message to the UI thread by calling `PostMessage`. `PostMessage` marshals and queues the message into the message queue associated with the thread that created the window handle. Do NOT use `SendMessage`, as `SendMessage` is run on the currently executing thread, which is not the correct (UI) thread.
- Test the C++ program to assure that the logic and thread management are correct.
- Declare a `cilk::context` and call the context's `run()` function (***Cilk++ runtime functions*** (Page 75)) from within this computation thread, creating the initial Cilk++ strand.
- Before terminating, the main (UI) thread should wait for the computation thread to complete, using `WaitForSingleObject()`.

The `QuickDemo` example illustrates a Cilk++ application using MFC.

Additional cautions:

- When the main UI thread creates the computation thread, it should not wait for the thread to complete. The function that creates the computation thread should return to allow the message pump to run.
- Be sure that none of the data passed to the computation thread is allocated on the stack. If it is, it will quickly be invalidated as the worker creation function returns, releasing the data.

- A data block passed to the computation thread should be freed by the computation thread when it is done, just before it sends a completion message to the UI.
- Use the `PostMessage` function instead of `CWnd::PostMessage`, as a primary reason for creating a computation thread is to avoid the MFC thread-local variables in Cilk++ code.

SHARED CILK++ LIBRARIES IN LINUX* OS

Create a shared library containing Cilk++ code in the same way as when creating a shared library containing C++ code, using the `-shared` compiler option.

Use the `cilk++` command to create the library. If you use a different command, use the linker option "`-z now`"; this option is automatically provided by the `cilk++` command. The `-z now` linker option disables lazy binding of functions. Setting the environment variable `LD_BIND_NOW` before running the program has the same effect. Lazy binding does not function in Cilk++ programs.

Examples:

```
cilk++ -shared -o libcilkstuff.so cilk1.o cilk2.o
g++ -shared -Wl,-z,now -shared -o libcilkstuff.so cilk1.o cilk2.o
```

You must link a small part of the runtime library into the main executable. The `cilk++` command will perform this step automatically. Otherwise, use the options:

```
-Wl,-z,now -lcilkrts_main -lcilkrts
```

when creating a program that calls Cilk++ code in a shared library.

Examples:

```
cilk++ -o program main.o -lcilk_library
g++ -o program -Wl,-z,now -lcilk_library \
    -lcilkrts_main -lcilkrts
```

CONVERTING WINDOWS* DLLS TO CILK++

The ***Getting Started*** (Page 8) section showed how to convert an existing C++ application to Cilk++. It's often necessary to convert a Windows dynamic link library (DLL) to Cilk++ without modifying the calling ("client") code in any way.

In particular, the programmer who uses a Cilk++ DLL should not need to change an existing C++ client application to get the performance gains from a converted library. The DLL might be provided by a third party, and the client application developer or user may not even know that the DLL is written in the Cilk++ language.

This section uses an example based on the previous `qsort` example to illustrate the conversion steps. The Microsoft Visual Studio* solution example, `qsort-dll`, contains three projects:

- `qsort-client` — the common C++ code to invoke the DLLs
- `qsort-cpp-dll` — the C++ quicksort implementation in a DLL
- `qsort-cilk-dll` — the Cilk++ quicksort implementation in a DLL

`qsort-client` is linked against both DLLs and will call one or the other based on the command line option (`-cpp` or `-cilk`). The Cilk++ DLL will be faster on a multicore system. Note that:

- `qsort-client` is written in C++ and is totally unaware of the fact that it calls a library written in Cilk++
- `qsort-client` does not use `cilk_main()`, and, therefore, you cannot use the –`cilk_set_worker_count` option.

Use the Cilk++ Compiler

Using the conversion process described in ***Getting Started*** (Page 8) as a guide, first convert the C++ code to use the Cilk++ compiler:

- Open the `qsort-cpp-dll` solution.
- Select and expand the `qsort-cpp-dll` project.
- Right click on the `qsort-cpp-dll.cpp` source file and select Convert to Cilk++.

Modify the C++ Source Code

Modify `qsort-dll-cpp.cpp` to create a Cilk++ context, and convert the `sample_qsort` call into a call that can be used by `cilk::context::run`. Here is a complete listing from the `qsort-dll-cilk` project, followed by an explanation of the changes.

```
/*
 * An DLL quicksort implementation in Cilk++.
 */
. . .
#include <windows.h>

#include <cilk.h>
#include "qsort.h"

static cilk::context ctx;

BOOL WINAPI DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved)
{
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            ctx.set_worker_count(0);  // Default count
            break;

        case DLL_PROCESS_DETACH:
            break;

        case DLL_THREAD_ATTACH:
            break;

        case DLL_THREAD_DETACH:
            break;

        default:
            break;
 56      }
 57
 58      return true;
 59  }
     . . .
 66  static void cilk_qsort(int * begin, int * end) {
         . . .
 77  }
 78
 79  // The cilk::context::run signature is different from
 80  // the cilk_qsort signature, so convert the arguments
 81  static int cilk_qsort_wrapper (void * args)
 82  {
 83      int *begin, *end;
 84      int **argi = (int **)args;
 85
 86      begin = argi[0];
 87      end   = argi[1];
 88
 89      cilk_qsort (begin, end);
 90
 91      return 0;
 92  }
 93
 94  extern "C++" void QSORT_DLL sample_cilk_qsort
                 (int * begin, int * end)
 95  {
 96      int * args[2] = {begin, end};
 97
 98      int retval =
             ctx.run (cilk_qsort_wrapper, (void *)args);
 99      return;
100  }
```

The code changes are:

- Include the Cilk++ header file: See Line 31.
- Create a global Cilk++ context variable (Line 34). This will construct the Cilk++ context when the DLL is loaded. We use `cilk::context::run`, rather than `cilk::run`, so that it is not necessary to create a context for every `sample_qsort()` call.
- Add a `DllMain` function: See Lines 36-59.
  - When the client process starts and first attaches to the DLL, set the number of workers, which, in this case, is the number of cores. Notice that there is no way to get the number of workers from the command line, as in the `qsort` example.
  - No action is necessary, in this limited example, when threads attach and detach.
- Rename the `qsort` function: See Line 81. This function has Cilk++ linkage, and it should have a different name from the `sample_qsort` function called by the client application (see Line 83).

- Modify the `sample_qsort` function: See Lines 94-100.
  - Specify C++ linkage since this is within a Cilk++ module (Line 94).
  - Copy the function arguments into a pointer array in order to conform to the `cilk::context::run` signature.
  - Call `cilk::context::run`, specifying a wrapper function, `cilk_sort_wrapper` with the correct signature that will convert the arguments to the form required by `cilk_qsort` (Line 66).
- Write `cilk_sort_wrapper`; see Lines 68-81. Convert the parameters to the form that `cilk_qsort` requires.

Note: Detecting Races in DLLs

The global Cilk++ context variable, `ctx` (Line 34) is constructed statically when `qsort-client` loads. Consequently, `cilkscreen` is not able to analyze the Cilk++ code in `cilk_qsort_wrapper()`, as `cilkscreen` starts after `qsort-client` and is not aware of `ctx`. There are no race conditions in this particular example, but, if there were, `cilkscreen` would not be able to detect them. The solution would be to call `cilk_qsort_wrapper()` from a `cilk_main()` test harness and run `cilkscreen` on that test harness before releasing the DLL for use by C++ programs.

Modify the DLL Header File

Change the header file, `qsort.h`, which is included by both the client and the DLL project, to give `sample_qsort` C++ linkage using `extern "C++"`. This has no impact on the client project but is required in the DLL project.

```
#ifndef _QSORT_H
#define _QSORT_H
#ifdef _WINDLL
#define QSORT_DLL __declspec(dllexport)
#else
#define QSORT_DLL __declspec(dllimport)
#endif

extern "C++"
    void QSORT_DLL sample_qsort(int * begin, int * end);
#endif
```

Chapter 10

Runtime System and Libraries

Cilk++ programs require the runtime system and libraries, which this chapter describes in four parts:

- `cilk::context` and its functions.
- `cilk::run`, which runs functions with Cilk++ linkage
- `cilk::mutex` and related objects
- The Miser memory manager

Include `cilk.h` to declare the Cilk++ runtime functions and classes. All runtime functions and classes, other than the Miser memory manager, are in the `cilk` namespace.

CILK::CONTEXT

A `cilk::context` is an object used to run Cilk++ functions from C++ code. C++ code cannot call Cilk++ functions directly because the languages use incompatible calling conventions.

`cilk::context` provides the following interface:

- `int run(void *fn, void *args)` runs a function, `fn`, with the specified arguments. `fn` must have Cilk++ linkage.
- `unsigned set_worker_count(unsigned n)` specifies the number of workers and returns the previous number of workers. By default, the Intel Cilk++ runtime system will create a worker for every physical core on the system. Processors with Hyper-Threaded technology are counted as a single processor.
  - Passing 0 resets the number of workers to the default.
  - `set_worker_count()` should not be called while Cilk++ code is running.
- `unsigned get_worker_count()` returns the number of workers.
- A constructor with no arguments.

The following two lines show how construct a context, start the runtime system, and execute a Cilk++ function from a C++ function:

```
cilk::context ctx;
ctx.run(cilk_function, (void *)&n);
```

The second parameter, (void *)&n, is an array of pointers to `cilk_function()` arguments.

The next section describes an alternative, the `cilk::run()` function, which does not require an explicit context and which takes an argument list.

In some cases, you will not need a context because your program does not need to call into the runtime system directly. `cilk_main()` is the standard Cilk++ program entry point, and it creates the context and initializes the runtime system. However, there are situations where it is appropriate to call Cilk++ code from C++ code, and the runtime is required. Examples include:

- `qsort-dll` (a Windows* example program), where a shared library (DLL) creates a context for each program that calls the library.
- In large programs when ***mixing C++ and Cilk++*** (Page 81).
- You must use C++ for the main program before executing Cilk++ code, as described in "***MFC with Cilk++ for Windows OS*** (Page 70)"

Getting the Worker Id

The runtime system provides one additional function that is not part of `cilk::context` but is convenient to describe here.

`int current_worker_id()` returns the worker ID for the worker currently executing the calling Cilk++ strand. A worker ID is a small integer. Each worker within a `cilk::context` has a unique worker ID. Note that this function is provided purely for informational purposes. No other runtime API accepts a worker ID. In general, Cilk++ code should not care which worker a strand is running on.

CILK::CURRENT_WORKER_COUNT

When running a `cilk_main()` program, there is no way to access the `context` that `cilk_main()` creates. Consequently, you cannot get or set the worker count or invoke `run()`. Since `cilk_main()` calls `run()` to execute the Cilk++ code, there is no need to call `run()` again. Also, the worker count cannot be set once Cilk++ code is running.

However, it may be useful to know the worker count, so the following function is in the `cilk` namespace.

```
unsigned cilk::current_worker_count()
```

Operation is as follows:

- If called from a Cilk++ worker, it will return the number of workers.
- If called outside of a Cilk++ worker, it will return 0.
- If called in serialized code, it will return 1.

Notice that the function name and behavior are different from the similar function:

```
cilk::context::get_worker_count()
```

CILK::RUN

`cilk::run()` runs Cilk++ functions and is an alternative to using `context::run` (in the `cilk` namespace). This function is easy to use as it lists the arguments directly, rather than assembling them in an array. Furthermore, there is no need to declare an explicit context.

The function must have Cilk++ linkage, such as:

```
extern "Cilk++" rettype function(argtype1, argtype2, argtype3,
... );
```

Note that:

- There can be up to 15 arguments.
- The return type, `rettype`, can be `void`.

Run `function` with `cilk::run()` as follows:

```
returnval = cilk::run(&function, arg1, arg2, arg3, ...);
```

The requirements are:

- The argument types must match so that `arg1` is compatible with `argtype1`, etc.
- `returnval` must be assignable (or constructable) from `retype`.
- If `retype` is `void`, no assignment would be possible; just call `cilk::run()`.
- To specify the number of workers, call `set_worker_count()` before calling `cilk::run()`.

For an example of `cilk::run()`, see `matrix-transpose`.

CILK::MUTEX AND RELATED FUNCTIONS

`cilk::mutex` objects provide the same functionality for Cilk++ strands as native OS locks (such as Windows* `CRITICAL_SECTION` and Pthreads `pthread_mutex_t` objects) provide for threads; they ensure that only one Cilk++ strand can lock the `cilk::mutex` at any time.

Mutexes are used primarily to remove ***data races*** (Page 123). The section on ***Locks*** (Page 92) describes potential problems with mutexes, such as ***deadlocks*** (Page 123) and determinacy races that are not data races.

Mutexes are not ordered. If multiple strands are waiting on a mutex and it becomes available, there is no way to predict which strand will be granted the mutex.

`cilk::mutex` is defined in `cilk_mutex.h` and provides the following three interface functions:

- `void lock()` waits until the mutex is available and then enters. Only one strand may enter the mutex at any time. There is no limit on how long the strand will wait to acquire the mutex.
- `void unlock()` releases the mutex for other strands to enter. It is an error to unlock a mutex that the strand has not locked.
- `bool try_lock()` returns `false` if the mutex is not available. If the mutex is available, the mutex is locked and the method returns `true`.

There are two additional objects related to mutexes:

- `cilk::fake_mutex` (defined in `fake_mutex.h`) is the equivalent of `cilk::mutex`, only it doesn't actually lock anything. Its sole purpose is to tell `cilkscreen` that it should consider some sequence of code protected by a "lock". The ***Race Condition*** (Page 125) chapter gives more information.

- `cilk::lock_guard` (defined in `lock_guard.h`) is an object that must be allocated on the stack. It calls the `lock` method on the `cilk::mutex` passed to its constructor, and it will call the `unlock` method on that mutex in its destructor. The destructor will be invoked automatically when the object goes out of scope. `cilk::lock_guard` is a template class. The template parameter is the type of mutex, which must have "`lock`" and "`unlock`" methods and default constructors/destructors. Specifically, you can use both `cilk::mutex` and `cilk::fake_mutex` , as well as any other class that has the required methods.

Note: A `cilk::mutex` is an example of a "lock", which is a more general term. The "***Locks and Their Implementation*** (Page 92)" section describes other locking mechanisms.

MISER MEMORY MANAGER

Some memory managers perform poorly when used by parallel applications, including Cilk++ programs. Therefore, the Intel® Cilk++ SDK provides an additional memory manager, "Miser", as a drop-in replacement for the system-provided C/C++ memory management functions (`new`, `delete`, `malloc`, `calloc`, `realloc`, and `free`).

Miser is transparent to the programmer; once it is enabled, C/C++ runtime memory management function calls are automatically forwarded to the Miser implementation.

Miser is *NOT* in the `cilk` namespace.

The following sections describe memory management limitations and the Miser solution. For in-depth discussion, see these articles:

- ***Multicore Storage Allocation*** (http://software.intel.com/en-us/articles/Multicore-Storage-Allocation) .
- ***Miser – A Dynamically Loadable Memory Allocator for Multithreaded Applications*** (http://software.intel.com/en-us/articles/Miser-A-Dynamically-Loadable-Memory-Allocator-for-Multi-Threaded-Applications) . This article includes a graph showing Miser's performance advantages using a solution to the N-Queens problem; without Miser, performance is best with just two cores. Using Miser, performance improves nearly linearly with the core count.

MEMORY MANAGEMENT LIMITATIONS

Some C/C++ runtime memory management functions, while thread safe, are optimized for performance and memory usage in a single threaded environment. The three principal limitations, two of which are caused by concurrency, are:

- Lock contention between strands (and worker threads) for access to the runtime memory management, which is globally locked. Lock contention can greatly reduce concurrency and performance.
- "***False sharing*** (Page 123)" is the situation where workers on different cores have memory allocated on the same cache line, which also reduces performance.
- Fragmentation caused by allocating and deallocating small memory blocks. This is a general problem not directly related to concurrency.

MISER MEMORY MANAGEMENT

Miser avoids these problems by combining several techniques:

- Miser avoids lock contention and false sharing by creating a distinct memory pool for each strand.
- Miser avoids fragmentation by rounding up allocation unit sizes to the nearest power of two for memory request sizes less than or equal to 256. This simplification improves performance, but does reduce memory allocation efficiency.
- Miser forwards allocation requests of size greater than 256 to the operating system allocator.

MISER INITIALIZATION

For ***Windows**** programmers only:

Enable Miser at runtime by loading the Miser DLL using the Windows `LoadLibrary` function, as follows:

```
#include <windows.h>
    . . .
HMODULE mdll = LoadLibrary ("Miser.dll");
if (NULL == mdll) {
    // Report and handle fatal error
}
```

Miser will handle all subsequent C/C++ runtime memory allocation calls in this program. There is no affect on other programs, including other Cilk++ programs.

Any operations on memory allocated before Miser was enabled, such as `free()` or `_msize()`, will be forwarded to the C/C++ runtime.

For ***Linux**** programmers only:

Miser is enabled at link time, not run time. To link with Miser, just use "`-lmiser`" on the command line. You can use Miser with C and C++ as well as Cilk++ programs.

Examples:

```
cilk++ -o myprog myprog.cilk -lmiser
gcc -o myprog myprog.c -lmiser
```

Alternatively, set the environment variable when executing the program (such as `a.out`) as follows, illustrated for 64-bit installation of the Intel® Cilk++ SDK to the default install location:

```
$ LD_PRELOAD=/usr/local/cilk/lib64/libmiser.so ./a.out
```

MISER LIMITATIONS

On all platforms:

- Once enabled, Miser cannot be disabled and will handle all memory management requests from C, C++ or CIlk++ code until the program terminates.

- Each Cilk++ program (process) enables Miser independently, and enabling Miser in one Cilk++ program does not affect memory management in another program.

On ***Windows**** platforms only:

- You cannot use the Windows `FreeLibrary()` function to free the `Miser.dll` module.
- The project must use the "Multithreaded DLL" compiler option: `/MD` or `/MDd`. Miser cannot intercept calls to the static C/C++ runtime library. The *Runtime Library* setting can be found on the *Code Generation* page of the *C/C++* compiler property pages in Microsoft Visual Studio*.
- Miser only affects the operation of the C RunTime Library memory management functions. It does not change the behavior of the system heap management functions, such as Windows `HeapCreate`, `HeapAlloc`, and `VirtualAlloc`.

Chapter 11

Mixing C++ and Cilk++ Code

A common problem is to add parallelism to a large C++ program without converting the entire program to the Cilk++ language. One approach is to convert entire classes to the Cilk++ language. There is a problem, however, since C++ code cannot call members of a Cilk++ class. Furthermore, C++ code cannot include any header file that declares Cilk++ functions without certain adjustments.

In some cases, the best strategy might be to start up the Cilk++ environment fairly close to the leaves of the call tree, using C++ wrappers to allow the Cilk++ functions to be callable from C++ (one of four approaches in the next section). The amount of work performed by each parallel function may be sufficient to offset the overhead of starting the runtime system each time through. The `qsort-dll` example, uses this approach.

The next sections describe techniques that allow C++ to call Cilk++ code.

In many cases, this is not necessary, and there is a fourth approach where Cilk++ code calls C++. Just convert the key loop or recursive function to the Cilk++ language. Then, call freely from Cilk++ code to C++ code, as long as the C++ code does not try to call back into Cilk++ code.

MIXING C++ AND CILK++: THREE APPROACHES

There are four general approaches to adding Cilk++ code to a project:

- Convert the entire project to the Cilk++ language
- Convert only the call tree leaves where the Cilk++ keywords are used, requiring that C++ functions call Cilk++ functions
- Some combination of these two approaches
- Structure the project so that Cilk++ code calls C++ code, but not conversely

Approach #1 may be too big a commitment for large projects, at least initially. The fourth approach may not be possible. Approach #2 can suffer from significant overhead of starting and stopping the Cilk++ environment on entry to each parallel function (although we are working to reduce that overhead in future releases). Approach #3 is a reasonable balance, but it is practical only when converting a module with a small number of public entry points.

For **Windows*** examples using the second method (convert the call tree leaves), see "***MFC with Cilk++ for Windows OS*** (Page 70)" (also, the `QuickDemo` example) and "***Converting Windows DLLs to Cilk++*** (Page 71)" (also, the `qsort-dll` example).

Approaches 2 and 3 both involve creating wrapper functions callable from C++ that start the Cilk++ environment and call a Cilk++ function using the `cilk::context::run` (Page 75) (or `cilk::run` (Page 76)) entry point. In the following code, the arguments (three in this case) are bundled into a single structure for use by the `run()` call.

Note that this code is C++, and the source files do not need to have the `.cilk` extension.

```
int myCilkEntrypoint (argType1 arg1, argType2 arg2,
                                     argType3 arg3)
{
    // Do parallel work
}

typedef struct
{
    argType1 arg1;
    argType2 arg2;
    argType3 arg3;
} argBundle;

int myCilkEntrypointWrapper (void *args)
{
    // Unbundle the parameters and do the work
    argBundle *data = (argBundle)args;
    return myCilkEntrypoint (data->arg1, data->arg2,
                                         data->arg3);
}

extern "C++"
int myCppEntrypoint (argType1 arg1, argType2 arg2,
                                    argType3 arg3)
{
    // Bundle parameters to call the Cilk++ entrypoint
    argBundle data = { arg1, arg2, arg2 };

    // Create Cilk++ context; call Cilk++ entrypoint,
    // passing it the argument bundle
    cilk::context ctx;
    ctx.run (myCilkEntrypointWrapper, (void *)&data);
}
```

If the function in question is a class member, then `myCppEntryPoint` and `myCilkEntryPointWrapper` will need to pack and unpack the `this` pointer as well as the arguments. All the code in the calling chain from the `cilk::context::run` call down to any function that uses `cilk_spawn`, `cilk_sync`, or `cilk_for` must have Cilk++ linkage. However, the Cilk++ functions can call back into C++ at the leaves of the call tree. Reduce the `run()` calling overhead by reusing the `cilk::context` object (e.g., by making it static).

HEADER FILE LAYOUT

Cilk++ functions, along with the `extern "Cilk++"` and `__cilk` keywords, should not be seen by the C++ compiler. It is necessary, therefore, to `#ifdef` some parts of the header file in order to share the header between files written in the Cilk++ language and those in C++. Use the `__cilkplusplus` predefined macro for this purpose. For example, given a class with a few parallel functions, the original class:

```
class MyClass
{
public:
    MyClass();
    int doSomethingSmall(ArgType);
    double doSomethingBig(ArgType1, ArgType2);
};
```

would be transformed into the something like the following if we wish to parallelize the `doSomethingBig` function:

```
extern "C++" {

class MyClass
{
#ifdef __cilkplusplus
private:
    // Parallel implementation of doSomethingBig
    double __cilk parallel_doSomethingBig(ArgType1, ArgType2);
    static double __cilk
            parallel_doSomethingBig_wrapper(void* args);
    void __cilk parallel_helper(ArgType2);
#endif
public:
    MyClass();
    int doSomethingSmall(ArgType);
    double doSomethingBig(ArgType1, ArgType2);
};

}
```

The `doSomethingBig` member function becomes a C++ wrapper around the function's parallel version, as described above. Note that the call to `cilk::context::run` requires a static (or global) function, hence the `parallel_doSomethingBig_wrapper` function to unpack the pointer. There is more information about these functions in the upcoming "***Source File Layout*** (Page 84)" section.

NESTED #INCLUDE STATEMENTS

Note that in the above example, we have kept most of the interface as C++ and added a small Cilk++ interface. In order to accomplish this, we needed to wrap essentially the entire header file in `extern "C++"`. This might cause problems when compiling certain headers (e.g., some Windows* ATL headers) using the C++ compiler. A function that is declared `extern "C"`, for example, could normally be defined without repeating the `extern "C"` specification. If the definition is wrapped in `extern "C++"`, however, the undecorated definition becomes illegal. The problem is limited to the native C++ compiler, not to the Cilk++ compiler.

Therefore, when compiling a header from within a C++ source file, it is desirable to remove the `extern "C++"` directives, which are at best redundant and at worse harmful. `cilk.h` provides a set of macros that evaluate to an `extern "C++"` region within a Cilk++ compilation and evaluate to nothing in a C++ compilation.

Use the following macros only for bracketing the `#include` of C++ header files, not as a general replacement for `extern "C++"`. Do not, for example, use the macros to start a C++ region within a C region, since they will be compiled out in a non-Cilk++ compilation.

```
#ifdef __cilkplusplus
# define CILK_BEGIN_CPLUSPLUS_HEADERS   extern "C++" {
# define CILK_END_CPLUSPLUS_HEADERS     }
#else
# define CILK_BEGIN_CPLUSPLUS_HEADERS
# define CILK_END_CPLUSPLUS_HEADERS
#endif
```

Use this as follows:

```
#include <cilk.h>

CILK_BEGIN_CPLUSPLUS_HEADERS
#include "MyHeaderFile.h"
CILK_END_CPLUSPLUS_HEADERS
```

These macros are part of `cilk.h`. See the earlier "***Calling C++ Functions from Cilk++*** (Page 46)" section for more about language linkage and these macros.

SOURCE FILE LAYOUT

For either Approach #2 or Approach #3, the original source file may contain a mixture of functions, some of which are to be converted to the Cilk++ language and others not. There are two ways to handle this:

- Create a separate `.cilk` file for the Cilk++ code.
- Convert the entire `.cpp` to `.cilk`, then wrap the C++ sections in `extern "C++"`, as shown below, using the macros defined in the previous section.

```
double MyClass::parallel_doSomethingBig(ArgType1, ArgType2)
{
    ...
    cilk_spawn parallel_helper(arg2);
    ...
}
double MyClass::parallel_doSomethingBig_wrapper(void* arg)
{
    ArgStruct* args = (ArgStruct*) args
    MyClass* self = args->self;
    ArgType1 arg1 = args->arg1;
    ArgType2 arg2 = args->arg2;
```

```
        return self->parallel_doSomethingBig(arg1, arg2);
    }
    void MyClass::parallel_helper(ArgType2) {...}

    extern "C++"
    {
    MyClass::MyClass() {...}
    int MyClass::doSomethingSmall(ArgType) {...}

    double MyClass::doSomethingBig(ArgType1 arg1, ArgType2 arg2)
    {
        ArgStruct args = { this, arg1, arg2 };
        cilk::context ctx;
        ctx.run(parallel_doSomethingBig, &args);
    }

    }  // extern "C++"
```

Since `extern "C++"` and `extern "Cilk++"` can nest, so you can also write:

```
    extern "C++"
    {
    // Other C++ code here

    extern "Cilk++" {
        double MyClass::parallel_doSomethingBig(ArgType1,
                                                ArgType2) {...}
        double MyClass::parallel_doSomethingBig_wrapper(void* arg)
               {...}
        void MyClass::parallel_helper(ArgType2) {...}
    } // end extern "Cilk++"

    MyClass::MyClass() {...}

    int MyClass::doSomethingSmall(ArgType) {...}

    double MyClass::doSomethingBig(ArgType1, ArgType2) {...}

    }  // extern "C++"
```

SERIALIZING MIXED C++/CILK++ PROGRAMS

A Cilk++ program often contains C++ modules. Those modules may need to use a few features from the Intel® Cilk++ SDK, such as reducers, without being completely converted into Cilk++. Conversion to Cilk++ means that the calling function must also be Cilk++, and so on up the call stack, which is inconvenient in many situations. For example, you may want to use reducers from within a C++ module without converting the entire module to Cilk++.

Supporting this usage model means that `cilk.h` cannot assume that, just because the compiler is a C++ compiler and not a Cilk++ compiler, that there is no Cilk++ runtime or that the user wants stubs for library facilities such as reducers. On the other hand, a true serialization of a Cilk++ program does require stubbing out the library facilities.

Resolve this issue by separating the concerns into two cases:

- A file that is used only when building the serialized program. That is, the compilation is for debugging or for use where the Cilk++ environment is not available.
- A C++ file that is part of a Cilk++ program is being compiled.

Cilk++ provides two header files to address these two cases:

- `cilk_stub.h` contains stubs for `cilk_spawn`, `cilk_sync`, and other language keywords. It will not contain definitions for interfaces to libraries provided with the Intel Cilk++ SDK. It also defines a macro, `CILK_STUB`.
- `cilk.h` contains the core Cilk++ library API. If `CILK_STUB` is defined, then inline stubs are provided for library interfaces where they make sense. Some features in `cilk.h` may be unavailable in a C++ compilation.

It should be rare that a source file would include `cilk_stub.h` directly. The `cilkpp` (**Windows* OS**) and `cilk++` (**Linux* OS**) wrappers will force inclusion of `cilk_stub.h` if serialization mode is turned on via the Windows compiler "`/cilkp cpp`" option and the "`-fcilk-stub`" Linux compiler option. Users who do not have a Cilk++ compiler available will be advised to force the inclusion of `cilk_stub.h` for any file that uses a Cilk++ language or library feature, whether that file be a `.cilk` or a `.cpp` file. Force inclusion with the Windows "`/FI`" and Linux "`-include`" options.

Chapter 12

Race Conditions

Races are a major cause of bugs in parallel programs. In this chapter, we describe what a race is, and programming techniques for avoiding or correcting race conditions. In the following chapter, we will describe how to use the cilkscreen race detector to find data races in your Cilk++ program.

For a more theoretical treatment, see ***What Are Race Conditions? Some Issues and Formalizations*** (http://portal.acm.org/citation.cfm?id=130616.130623), by Robert Netzer and Barton Miller. Note that the paper uses the term *general race* where we would say *determinacy race*. A *data race* is a special case of determinacy race.

DATA RACES

A *data race* occurs when two parallel strands, holding no locks in common, access the same memory location and at least one strand performs a write. The program result depends on which strand "wins the race" and accesses the memory first.

For example, consider the following very simple program:

```
int a = 2;    // declare a variable that is
              // visible to more than one worker

void Strand1()
{
    a = 1;
}

int Strand2()
{
    return a;
}

void Strand3()
{
    a = 2;
}

int cilk_main()
{
    int result;

    cilk_spawn Strand1();
    result = cilk_spawn Strand2();
    cilk_spawn Strand3();
    cilk_sync;
```

```
        std::cout << "a = " << a << ", result = "
                  << result << std:endl;
    }
```

Because `Strand1()`, `Strand2()` and `Strand3()` may run in parallel, the final value of a and result can vary depending on the order in which they run.

`Strand1()` may write the value of "a" before or after `Strand2()` reads "a", so there is a *read/write race* between `Strand1()` and `Strand2()`.

`Strand3()` may write the value of "a" before or after `Strand1()` writes "a", so there is a *write/write race* between `Strand3()` and `Strand1()`.

DETERMINACY RACES

A *determinacy race* occurs when two parallel strands access the same memory location and at least one strand performs a write. The program result depends on which strand "wins the race" and accesses the memory first.

Observe that a data race is a special case of a determinacy race.

If the parallel accesses are protected by locks, then by our definition, there is no data race. However, a program using locks may not produce deterministic results. A lock can ensure consistency by protecting a data structure from being visible in an intermediate state during an update, but does not guarantee deterministic results. More details are provided when we discuss ***locks and races***.

BENIGN RACES

Some data races are benign. In other words, although there is a race, you can prove that the race does not affect the output of the program.

Here is a simple example:

```
bool bFlag = false;
cilk_for (int i=0; i<N; ++i)
{
    if (some_condition()) bFlag = true;
}
if (bFlag) do_something();
```

This program has a write/write race on the `bFlag` variable. However, all of the writes are writing the same value (`true`) and the value is not read until after the `cilk_sync` that is implicit at the end of the `cilk_for` loop.

In this example, the data race is benign. No matter what order the loop iterations execute, the program will produce the same result.

RESOLVING DATA RACES

There are several ways to resolve a race condition:

- Fix a bug in your program
- Use local variables instead of global variables
- Restructure your code
- Change your algorithm
- Use reducers
- Use a Lock

Here is a brief description of each of these techniques:

Fix a bug in your program

The race condition in `qsort-race` is a bug in the program logic. The race is caused because the recursive sort calls use an overlapping region, and thus reference the same memory location in parallel. The solution is to fix the application.

Use local variables instead of global variables

Consider the following program:

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

const int IMAX=5;
const int JMAX=5;
int a[IMAX * JMAX];

int cilk_main()
{
    int idx;

    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            idx = i*JMAX + j;          // This is a race.
            a[idx] = i+j;
        }
    }

    for (int i=0; i<IMAX*JMAX; ++i)
        std::cout << i << " " << a[i] << std::endl;
    return 0;
}
```

This program has a race on the `idx` variable, because it is accessed in parallel in the `cilk_for` loop. Because `idx` is only used inside the lop, it is simple to resolve the race by making `idx` local within the loop:

```
int cilk_main()
{
//  int idx;                           // Remove global
    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            int idx = i*JMAX + j;      // Declare idx locally
            a[idx] = i+j;
        }
    }
}
```

Restructure your code

In some cases, you can eliminate the race by a simple rewrite. Here is another way to resolve the race in the previous program:

```
int cilk_main()
{
//  int idx;                           // Remove global
    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
//          idx = i*JMAX + j;          // Don't use idx
            a[i*JMAX + j] = i+j;       // Instead,
                                       // calculate as needed
        }
    }
}
```

Change your algorithm

One of the best solutions, though not always easy or even possible, is to find an algorithm that partitions your problem such that the parallelism is restricted to calculations that cannot race. A detailed description of this approach is described in ***a solution to N-body interactions*** http://software.intel.com/en-us/articles/A-cute-technique-for-avoiding-certain-race-conditions.

Use reducers

Reducers are designed to be race-free objects that can be safely used in parallel. See the ***chapter on Reducers*** (Page 52) for more information.

Use locks

Locks can be used to resolve data race conditions, albeit with the drawbacks described in the ***chapter on Locks*** (Page 92). There are several kinds of locks, including:

- cilk::mutex objects
- System locks on Window*s or Linux* systems
- Atomic instructions that are effectively short-lived locks that protect a read-modify-write sequence of instructions

The following simple program has a race on sum, because the statement sum=sum+i both reads and writes sum:

```
int cilk_main()
{
    int sum = 0;
    cilk_for (int i=0; i<10; ++i)
    {
        sum = sum + i;                  // THERE IS A RACE ON SUM
    }
}
```

Using a lock to resolve the race:

```
#include <cilk_mutex.h>

int cilk_main()
{
        cilk::mutex mut;
        int sum = 0;
        cilk_for (int i=0; i<10; ++i)
        {
                mut.lock();
                sum = sum + i;          // PROTECTED WITH LOCK
                mut.unlock();
        }
        std::cout << "Sum is " << sum << std::endl;

        return 0;
}
```

Note that this is for illustration only. A reducer is typically a better way to solve this kind of race.

Chapter 13

Locks

There are many synchronization mechanisms that may be implemented in the hardware or operating system.

The race detector, described in detail in the next chapter, recognizes the following locking mechanisms; it does not recognize any others.

- The Intel® Cilk++ SDK provides the `cilk::mutex` to create critical code sections where it is safe to update and access shared memory or other shared resources safely. `cilkscreen` recognizes the lock and will not report a race on a memory access protected by the `cilk::mutex`. The `qsort-mutex` example shows how to use a `cilk::mutex`.
- **Windows* OS: *Windows*** `CRITICAL_SECTION` objects provide nearly the same functionality as `cilk::mutex` objects. `cilkscreen` will not report races on accesses protected by `EnterCriticalSection()`, `TryEnterCriticalSection()`, or `LeaveCriticalSection()`.
- **Linux* OS: *Posix* Pthreads mutexes*** (`pthead_mutex_t`) provide nearly the same functionality as `cilk::mutex` objects. `cilkscreen` will not report races on accesses protected by `pthread_mutex_lock()`, `pthread_mutex_trylock()`, or `pthread_mutex_unlock()`.
- `cilkscreen` recognizes atomic hardware instructions, available to C++ programmers through compiler intrinsics.

There are other operating system-specific mutexes, but these methods are nearly always slower than a `cilk::mutex`. Furthermore, `cilkscreen` will not recognize the other mutexes and could report a data race that does not exist.

Several basic lock terms and facts are useful:

- We speak interchangeably of "acquiring", "entering", or "locking" a lock (or "mutex").
- A strand (or thread) that acquires a lock is said to "own" the lock.
- Only the owning strand can "release", "leave", or "unlock" the lock.
- Only one strand can own a lock at a time.
- `cilk::mutex` is implemented using Windows `CRITICAL_SECTION or Linux pthread_mutex_t` objects.

Lock contention can create performance problems in parallel programs. Furthermore, while locks can resolve data races, programs using locks are often non-deterministic. We recommend avoiding locks whenever possible.

These problems (and others) are described in detail in the following sections.

LOCKS CAUSE DETERMINACY RACES

Even though you properly use a lock to protect a resource (such as a simple variable or a list or other data structure), the actual order that two strands modify the resource is not deterministic. For example, suppose the following code fragment is part of a function that is spawned, so that several strands may be executing the code in parallel.

```
. . .
// Update is a function that modifies a global variable, gv.
sm.lock();
Update(gv);
sm.unlock();
. . .
```

Multiple strands will race to acquire the lock, `sm`, so the order in which `gv` is updated will vary from one program execution to the next, even with identical program input. This is the source of non-determinism, but it is not a data race by the definition:

> A data race is a race condition that occurs when two parallel strands, *holding no locks in common*, access the same memory location and at least one strand performs a write.

This non-determinacy may not cause a different final result if the update is a ***commutative operation*** (Page 122), such as integer addition. However, many common operations, such as appending an element to the end of a list, are not commutative, and so the result varies based on the order in which the lock is acquired.

DEADLOCKS

A deadlock can occur when using two or more locks and different strands acquire the locks in different orders. It is possible for two or more strands to become deadlocked when each strand acquires a mutex that the other strand attempts to acquire.

Here is a simple example, with two strand fragments, where we want to move a list element from one list to another in such a way that the element is always in *exactly one* of the two lists. `L1` and `L2` are the two lists, and `sm1` and `sm2` are two `cilk::mutex` objects, protecting `L1` and `L2`, respectively.

```
// Code Fragment A. Move the beginning of L1 to the end of L2.
sm1.lock();
sm2.lock();
L2.push_back(*L1.begin);
L1.pop_front();
sm2.unlock();
sm1.unlock();
    ...
    ...
// Code Fragment B. Move the beginning of L2 to the end of L1.
sm2.lock();
sm1.lock();
L1.push_back(*L2.begin);
L2.pop_front();
```

```
    sm2.unlock();
    sm1.unlock();
```

The deadlock would occur if one strand, executing Fragment A, were to lock `sm1` and, in another strand, Fragment B were to lock `sm2` before Fragment A locks `sm2`. Neither strand could proceed.

The common solution for this example is to acquire the locks in exactly the same order in both fragments; for example, switch the first two lines in Fragment B. A common practice is to release the locks in the opposite order, but doing so is not necessary to avoid deadlocks.

There are ***extensive references*** (http://en.wikipedia.org/wiki/Deadlock) about deadlocks and techniques to avoid them.

LOCKS CONTENTION REDUCES PARALLELISM

Parallel strands will not be able to run in parallel if they concurrently attempt to access a shared lock. In some programs, locks can eliminate virtually all of the performance benefit of parallelism. In extreme cases, such programs can even run significantly *slower* than the corresponding single-processor serial program. Consider using a reducer if possible.

Nonetheless, if you must use locks, here are some guidelines.

- Hold a synchronization object (lock) for as short a time as possible (but no shorter!). Acquire the lock, update the data, and release the lock. Do not perform extraneous operations while holding the lock. If the application must hold a synchronization object for a long time, then reconsider whether it is a good candidate for parallelization. This guideline also helps to assure that the acquiring strand always releases the lock.
- Always release a lock at the same scope level as it was acquired. Separating the acquisition and release of a synchronization object obfuscates the duration that the object is being held, and can lead to failure to release a synchronization object and deadlocks. This guideline also assures that the acquiring strand also releases the lock.
- Never hold a lock across a `cilk_spawn` or `cilk_sync` boundary. This includes across a `cilk_for` loop. See the following section for more explanation.
- Avoid deadlocks by assuring that a lock sequence is always acquired in the same order. Releasing the locks in the opposite order is not necessary but can improve performance.

HOLDING A LOCK ACROSS A STRAND BOUNDARY

The best and easiest practice is to avoid holding a lock across strand boundaries. Sibling strands can use the same lock, but there are potential problems if a parent shares a lock with a child strand. The issues are:

- The cilkscreen race detector assumes that everything protected by a synchronization object is protected from racing. So spawning a child function while holding a lock prevents the race detector from considering whether there are races between the two strands.
- There is no guarantee that a strand created after a `cilk_spawn` or `cilk_sync` boundary will continue to execute on the same OS thread as the parent strand. Most locking synchronization objects, such as a Windows* `CRITICAL_SECTION`, must be released on the same thread that allocated them.

- `cilk_sync` exposes the application to Cilk++ runtime synchronization objects. These can interact with the application in unexpected ways. Consider the following code:

```
int child (cilk::mutex &m, int &a)
{
    m.lock();
    a++;
    m.unlock();
}

int parent(int a, int b, int c)
{
    cilk::mutex m;
    try
    {
        cilk_spawn child (&m, a);
        m.lock();
        throw a;
    }
    catch (...)
    {
        m.unlock();
    }
}
```

There is an implied `cilk_sync` at the end of a try block which contains a `cilk_spawn`. In the event of an exception, execution cannot continue until all children have completed. If the parent acquires the lock before a child, the application is deadlocked since the catch block cannot be executed until all children have completed, and the child cannot complete until it acquires the lock. Using a "guard" object won't help, because the guard object's destructor won't run until the catch block is exited.

To make the situation worse, invisible try blocks are everywhere. Any compound statement that declares local variables with non-trivial destructors has an implicit try block around it. Thus, by the time the program spawns or acquires a lock, it is probably already in a try block.

The rule, then, is: if a function holds a lock that could be acquired by a child, the function should not do anything that might throw an exception before it releases the lock. However, since most functions cannot guarantee that they won't throw an exception, follow these rules:

- Do not acquire a lock that might be acquired by a child strand. That is, lock against your siblings, but not against your children.
- If you need to lock against a child, put the code that acquires the lock, performs the work, and releases the lock into a separate function and call it rather than putting the code in the same function as the spawn.
- If a parent strand needs to acquire a lock, set the values of one or more primitive types, perhaps within a data structure, then release the lock. This is always safe, provided there are no try blocks, function calls that may throw (including overloaded operators), spawns or syncs involved while holding the lock. Be sure to pre-compute the primitive values before acquiring the lock.

Chapter 14

Cilkscreen Race Detector

The cilkscreen race detector monitors the actual operation of a Cilk++ program as run with some test input. `cilkscreen` reports all ***data races*** (Page 123) encountered during execution. By monitoring program execution, `cilkscreen` 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.

In order to ensure a reliable parallel program, you should review and resolve all races that `cilkscreen` reports. You can instruct `cilkscreen` to ignore benign races.

`cilkscreen` runs your program on a single worker and monitors all memory reads and writes. When the program terminates, `cilkscreen` outputs 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 `cilkscreen` on a Cilk++ 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. `cilkscreen` can only analyze code that is executed, and races may only occur under certain data and execution path combinations.

Please note that `cilkscreen` only detects and reports races that result from parallelism created by Cilk++ keywords. `cilkscreen` will NOT report races that result from conflicts between threads created using other parallel libraries or created explicitly using system calls.

`cilkscreen` recognizes and correctly manages many special cases, including:

- Cilk++ keywords
- Calls into the Cilk++ runtime
- Reducers
- `cilk::mutex` locks and some operating system locks
- Miser memory allocation
- Operating system memory management calls (`malloc` and others)
- Program calls to control `cilkscreen`

Because `cilkscreen` understands the parallel structure of your program (that is, which strands can run in parallel and which always run in series), the algorithm is provably correct. For more information, see "***Additional Resources and Information*** (Page 7)").

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

USING CILKSCREEN

Run `cilkscreen` from the command line:

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

Within Visual Studio*, select `Run Cilkscreen Race Detector` from the `Tools` menu. `cilkscreen` runs the program using the command arguments specified in the debugging property of the project. Remember that the `cilkscreen` menu items are not available in Visual C++ 2008 Express Edition.

You can run production or debug versions of your program under `cilkscreen`. 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 Cilk++ program.

To ensure high reliability, we recommend that you run `cilkscreen` 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 `cilkscreen` 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 races 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 `cilkscreen` to see the race reports.

- `qsort-race`, one of the Quicksort Examples. The race exists because two spawned recursive function calls use overlapping data ranges.
- ***Using Reducers — a Simple Example*** (Page 53), in which multiple strands, running in parallel, update a single shared variable. A `cilk_for` statement creates the strands. The example is then modified to use a reducer, eliminating the data race.

Command Line Options

`cilkscreen` recognizes the following command line options. These options are not available when running `cilkscreen` from within Visual Studio.

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

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

`-r reportfile`

Write output in ASCII format to report`file`.

Windows GUI applications do not have `stderr`, so output will be lost unless you use the `-r` option.

`-x xmlfile`

Write `cilkscreen` 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 ASCII trace 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 `cilkscreen` process.

`-s`

Display the command passed to PIN. PIN is the dynamic instrumentation package that `cilkscreen` uses to monitor instructions.

`-h`, `-?`

Display `cilkscreen` usage and exit.

`-v`

Display version information and exit.

`-w`

Run the ***Cilkscreen Parallel Performance Analyzer*** (Page 105).

UNDERSTANDING CILKSCREEN OUTPUT

Consider the following program `sum.cilk`:

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

int sum = 0;

void f(int arg)
{
    sum += arg;
}

int cilk_main()
{
    cilk_spawn f(1);
    cilk_spawn f(2);

    cilk_sync;
    std::cout << "Sum is " << sum << std::endl;
18        return 0;
19    }
```

Build the program with debugging information and run `cilkscreen`

- for Windows* OS:

  ```
  cilkpp /Zi sum.cilk
  cilkscreen sum.exe
  ```

- for Linux* OS:

  ```
  cilk++ -g sum.cilk
  cilkscreen sum
  ```

On Windows, `cilkscreen` generates output similar to this:

```
Race condition on location 004367C8
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
  read access at 004268CF: (c:\sum.cilk:8, sum.exe!f+0x11)
    called by 004269B4: (c:\sum.cilk:14, sum.exe!cilk_main+0xd0)
    called by 0042ABED: (c:\program files\microsoft visual studio
8\vc\include\ostream:786, sum.exe!__cilk_main0+0x3d)
    called by 100081D5: (cilk_1_1-
x86.dll!__cilkrts_ltq_overflow+0x137)
Variable: 004367C8 - int sum

Race condition on location 004367C8
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
    called by 004269B4: (c:\sum.cilk:14, sum.exe!cilk_main+0xd0)
    called by 0042ABED: (c:\program files\microsoft visual studio
8\vc\include\ostream:786, sum.exe!__cilk_main0+0x3d)
    called by 100081D5: (cilk_1_1-
x86.dll!__cilkrts_ltq_overflow+0x137)
Variable: 004367C8 - int sum
Sum is 3
2 errors found by Cilkscreen
Cilkscreen suppressed 1 duplicate error messages
```

Here is what the output means:

There is a race condition detected at memory location `004367C8`.

```
Race condition on location 004367C8
```

The first access that participated in the race was a write access from location `004268D8` in the program. This co
`sum.cilk`, from the instruction at offset `0x1a` in the function `f()` loaded from the executable binary file `sum.e`
that the source code at line 8 is

```
sum += arg;
```

```
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
```

The second access was a read from location 004268CF in the program. This also corresponds to source line 8 at offset 0x11 in f(). The write access occurs when storing the new value back to sum. The read access occur memory in order to add the value of arg to it.

```
    read access at 004268CF: (c:\sum.cilk:8, sum.exe!f+0x11)
```

cilkscreen displays a stack trace for the second memory reference involved in the race. It would be very exp the stack trace for every memory reference. It is much cheaper to record the stack trace only when a race is det at line 14 in sum.cilk, called from offset 0xd0 in the file cilk_main. We can see in the source code that lin

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

```
      called by 004269B4: (c:\sum.cilk:14, sum.exe!cilk_main+0xd0)
```

The next two lines in the stack trace show that cilk_main() was called from an initializer in ostream, which the Cilk++ runtime system that was loaded from cilk_1_1-x86 dll.

```
      called by 0042ABED: (c:\program files\microsoft visual studio 8\vc\include\
sum.exe!__cilk_main0+0x3d)
      called by 100081D5: (cilk_1_1-x86.dll!__cilkrts_ltq_overflow+0x137)
```

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

```
Variable: 004367C8 - int sum
```

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

```
Race condition on location 004367C8
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
  write access at 004268D8: (c:\sum.cilk:8, sum.exe!f+0x1a)
    called by 004269B4: (c:\sum.cilk:14, sum.exe!cilk_main+0xd0)
    called by 0042ABED: (c:\program files\microsoft visual studio 8\vc\include\
sum.exe!__cilk_main0+0x3d)
    called by 100081D5: (cilk_1_1-x86.dll!__cilkrts_ltq_overflow+0x137)
Variable: 004367C8 - int sum
```

Next we see the output produced by the program itself:

```
Sum is 3
```

cilkscreen summarizes the results. There were two distinct races reported. A third race was a duplicate of o output was suppressed.

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

CONTROLLING CILKSCREEN FROM A CILK++ PROGRAM

For the advanced user, `cilkscreen` provides a set of *pseudo-calls* that you can issue from your program. We describe these as pseudo-calls because they have the appearance of function calls in the source, but the Cilk++ compiler removes the calls and uses ***metadata*** to implement the requested operation. This mechanism allows you to customize `cilkscreen` from within your program without measurable cost when you do not run your program under `cilkscreen.` These pseudo-functions are all declared in `cilk.h` or `fake_mutex.h.`

Disable/Enable Instrumentation

`cilkscreen` begins instrumenting your program when you enter the Cilk++ context, either through a call to `cilk::run`, or implicitly before `cilk_main()` is called. 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 `cilkscreen` 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 `cilk_main()` before any parallel constructs are invoked.

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

For example:

```
int cilk_main()
{
    __cilkscreen_disable_instrumentation();
    very_expensive_initialization();
    __cilkscreen_enable_instrumentation();

    parallel_program();

    return 0;
}
```

Fake Locks

If you are confident that a race in your program is ***benign***, you can suppress the `cilkscreen` race report by using a *fake 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.

```
cilk::fake_mutex fake;
fake.lock();
fake.unlock();
```

Do not forget to call `unlock()` in all paths through your code! Alternately, you can use a `cilk::lock_guard` object, which will automatically release the lock from within the destructor that is called when the object goes out of scope.

For example, `cilkscreen` will not report a race in this program:

```
bool isUpdated = false;
cilk::fake_mutex fakeUpdateLock;

void f(int arg)
{
    fakeUpdateLock.lock();
    isUpdated = true;
    fakeUpdateLock.unlock();
}

int cilk_main()
{
    cilk_spawn f(1);
    cilk_spawn f(2);
    cilk_sync;

    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, `cilkscreen` continues to instrument your program with all of the corresponding overhead. While checking is disabled, `cilkscreen` will not record any memory accesses.

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

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

```
int sum = 0;

void f(int arg)
{
    sum += arg;
}

int cilk_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.

Cleaning Memory

`cilkscreen` needs to recognize when memory is allocated and freed. Otherwise, `cilkscreen` 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.

`cilkscreen` 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 `cilkscreen` when memory should be considered "clean" and thus any previous accesses to that memory should be forgotten.

```
    void __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;
    }
```

CILKSCREEN PERFORMANCE

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

In order to keep track of all memory references, `cilkscreen` 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 `cilkscreen` .

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

- `cilkscreen` 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.
- `cilkscreen` runs the program on a single worker, so there is no multicore speedup.
- `cilkscreen` forces the `cilk_for` grainsize to 1 in order to ensure that all races between iterations are detected.
- `cilkscreen` 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 `cilkscreen` recognizes various aspects of your program such as memory allocation calls, program calls to `cilkscreen` , and the Cilk++ keywords. You may wonder whether the performance of your production program is slowed down in order to support this. In fact, `cilkscreen` 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, `cilkscreen` calls add virtually no overhead when your program is not run under `cilkscreen` .

Chapter 15

Cilkview Scalability Analyzer

The cilkview scalability and performance analyzer is designed to help you understand the parallel performance of your Cilk++ program. `cilkview` can:

- Report parallel statistics about a Cilk++ program
- Predict how the performance of a Cilk++ program will scale on multiple processors
- Automatically benchmark a Cilk++ program on one or more processors
- Present performance and scalability data graphically

Similar to `cilkscreen`, `cilkview` monitors a binary Cilk++ application. However, `cilkview` does not monitor and record all memory accesses, and so incurs much less run time.

In this chapter, we will illustrate how to use `cilkview` with a sample Cilk++ 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.

CILKVIEW ASSUMPTIONS

Like `cilkscreen`, `cilkview` monitors your program as it runs on a single worker. In order to estimate performance on multiple cores, `cilkview` makes some assumptions:

- If no explicit grain size is set, `cilkview` analyzes `cilk_for` loops with a granularity of 1. This may lead to misleading results. You can use the `cilk_grainsize` pragma to force `cilkview` 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 `cilkview` does not indicate problems but your program uses reducers and runs slowly on multiple workers, review the section on ***Reducer performance considerations***.
- `cilkview` analyzes a single run of your Cilk++ program with a specific input data set. Performance will generally vary with different input data.

RUNNING CILKVIEW

Run `cilkview` from the command line:

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

The resulting report looks something like this:

```
Cilkscreen Scalability Analyzer V1.1.0, Build 8223
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
```

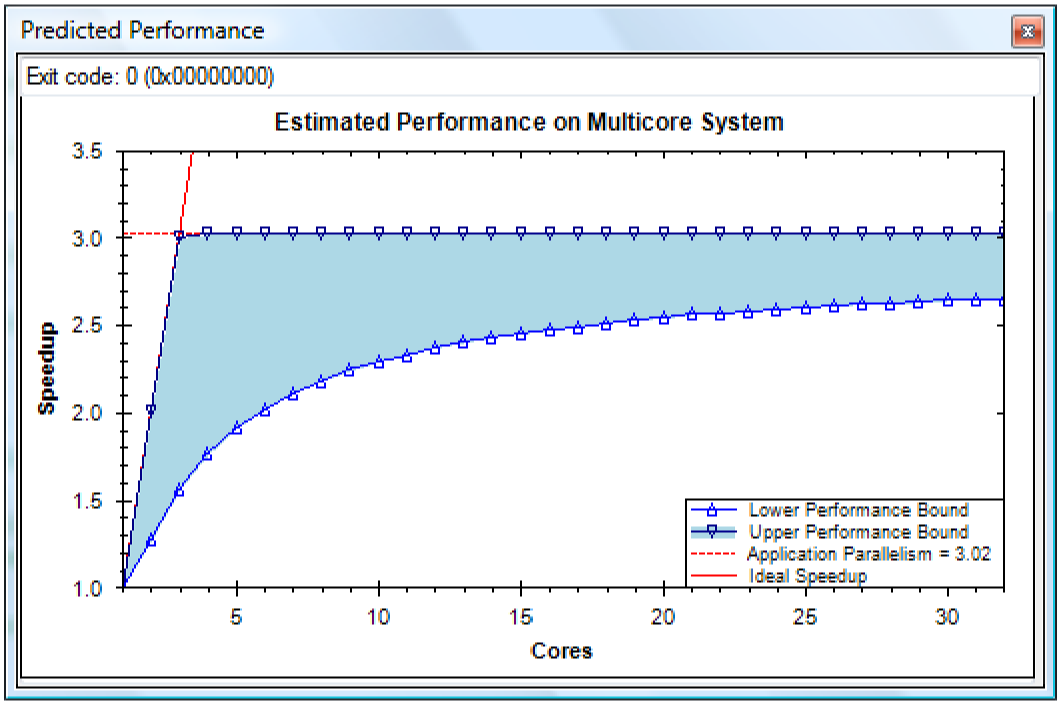
The exact number will vary depending on the program input, which compiler is used, and the compiler options selected.

The next section describes what the various numbers mean.

Windows* OS: You can also run `cilkview` from within Visual Studio* (not available with Visual C++ 2008 Express Edition):

- Open and build the Cilk++ project.
- Select **Tools -> Run Cilkview Scalability Analyzer** to run the program under `cilkview` with the command arguments specified in the project debugging properties.

`cilkview` 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.

`cilkview` writes output into a `.csv` file. Select **Tools -> Open Cilkview Log** to view the graph from a previously created `.csv` log file.

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.

WHAT THE PROFILE NUMBERS MEAN

The `cilkview` 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. In order to get a complete picture of the parallel operation of the program, `cilkview` 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

`cilkview` 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 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 Cilk++, 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 gransize.

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 Cilk++ 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. See the section on ***Locks*** (Page 92) for more information.

CILKVIEW EXAMPLE

In this section, we present a sample Cilk++ program that has a parallel performance issue, and show how `cilkview` 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 `cilkview` 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 is 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 `cilkview` 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 `cilkview` reports, the parallelism and burdened parallelism have both improved to 4.00.

ANALYZING PORTIONS OF A CILK++ PROGRAM

By default, `cilkview` 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 `cilkview` API allows you to control which portions of your Cilk++ program are analyzed. The API is accessed through methods on a `cilkview` object as defined in `cilkview.h`.

If you use the `cilkview` API, you must link with the `cilkutil` library. On Windows* systems, the appropriate library file (`cilkutil-x86.lib`) is linked automatically. On Linux* systems, use `-lcilkutil`:

```
>cilk++ program.cilk -lcilkutil
```

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

```
#include "cilk.h"
#include "cilkview.h"

cilk::cilkview cv;           // create a cilkview object

int cilk_main ()
{
   cv.start();                    // begin analyzing
   do_some_stuff();               // execute some parallel work
   cv.stop();                // stop analyzing
   cv.dump("main_tag");      // write analysis to file

   return 1;
}
```

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

```
   cv.start();                    // begin analyzing
   do_first_stuff();              // execute some parallel work
   cv.stop();                // stop analyzing
   cv.dump("first_tag");          // write analysis to file

   cv.start();                    // analyze another parallel
region
   do_different_stuff();          // execute more parallel work
   cv.stop();
   cv.dump("second_tag");
```

You can also use the same tag repeatedly. In this case, `cilkview` 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)
{
   cv.start();
   do_parallel_stuff();
   cv.stop();
   cv.dump("stuff_tag");
}
```

BENCHMARKING A CILK++ PROGRAM

In addition to parallel analysis, `cilkview` also provides a framework for benchmarking a Cilk++ program on various numbers of processors. A benchmark run is a complete run of your Cilk++ program using a specified number of workers. `cilkview` 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.

CILKVIEW REFERENCE

`cilkview` Output

`cilkview` prints performance statistics to `stdout`. When the target program terminates, `cilkview` prints statistics representing the entire program run. In addition, every time a Cilk++ program calls the `dump(TAGNAME)` method, intermediate data is printed to `stderr` and written to the `TAGNAME` file for plotting.

`cilkview` creates the following files:

`prog.cv.out`	Raw performance data collected for `prog` by `cilkview` .
`TAGNAME.csv`	Comma Separated Value file that can be read by most plotting/charting programs, including `cilkplot`.
`TAGNAME.plt`	Data files that can be read by `gnuplot`.

`cilkview` Command Line Options

The `cilkview` command line has the form:

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

`program [program arguments]`	Specify the Cilk++ program to run, and any arguments to that program.

`[options]`	Zero or more of the following:
`-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 `cilkview` is run on a Linux* system.
`-plot cilkplot`	Run `cilkplot` to display the output (Windows* systems only). This is the default behavior when `cilkview` is run on a Windows system. Cilkplot is distributed with the Intel® Cilk++ SDK.
`-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 usage information and exit.

`cilkview` and Visual Studio*

At this time, `cilkview` 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.

`cilkview` Application Program Interface (API)

In order for `cilkview` to measure the parallel performance of your program, you must create a `cilk::cilkview` object and call methods on that object to indicate the region of the program to measure. To use the `cilk::cilkview` object:

`#include <cilkview.h>`	Bring in the definition of `cilkview.`
`cilk::cilkview cv;`	Create an instance of a `cilkview` object named `cv`.
`cv.start();`	Begin measuring parallel performance.
`cv.stop();`	Stop measuring parallel performance.
`cv.dump("tagname", bReset);`	Dump the data collected so far and tag it with the specified character string. This string must not have white space or any characters that are not allowed in file names. If you call `dump()` more than once with the same tag, `cilkview` treats the data as representing multiple runs of the same program or program region, and selects the shortest (least time) data set for analysis and display. If `bReset` is `true` (or not specified), then `dump()` will reset data collection.
`cv.reset();`	Reset data collection. If data collection is not reset, data continues to accumulate.
`cv.accumulated_milliseconds();`	Return the number of milliseconds accumulated between `start()` and `stop()`. Many of the Cilk++ examples use this to print the elapsed time of a parallel region.

View Data with Cilkplot

The Cilkplot 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 `cilkview` . This allows you to easily view a plot of `cilkview` data without re-running the measurements.

View Data with gnuplot

`cilkview` 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 `cilkview.` This allows you to easily view a plot of `cilkview` data without re-running the measurements.

Chapter 16

Performance Issues in Cilk++ Programs

The previous chapter showed how to use `cilkview` to increase and exploit program parallelism. However, parallel programs have numerous additional performance considerations and opportunities for tuning and improvement.

In general, the Intel Cilk++ runtime scheduler uses processor resources efficiently. The work-stealing algorithm is designed to minimize the number of times that work is moved from one processor to another.

Additionally, the scheduler ensures that space use is bounded linearly by the number of workers. In other words, a Cilk++ program running on N workers will use no more than N times the amount of memory that is used when running on one worker.

While program performance is a large topic and the subject of numerous papers and books, this chapter describes some of the more common issues seen in multithreaded applications such as those written in Cilk++.

GRANULARITY

Divide-and-conquer is an effective parallelization strategy, creating a good mix of large and small sub-problems. The work-stealing scheduler can allocate chunks of work efficiently to the cores, provided that there are not too many very large chunks or too many very small chunks.

- If the work is divided into just a few large chunks, there may not be enough parallelism to keep all the cores busy.
- If the chunks are too small, then scheduling overhead may overwhelm the advantages of parallelism. The Intel® Cilk++ SDK provides `cilkview` to help determine if the chunk size (or granularity) is optimal.

Granularity can be an issue with parallel programs using `cilk_for` or `cilk_spawn`. If you are using `cilk_for`, you can control the granularity by ***setting the grain size*** of the loop. In addition, if you have nested loops, the nature of your computation will determine whether you will get best performance using `cilk_for` for inner or outer loops, or both. If you are using `cilk_spawn`, be careful not to spawn very small chunks of work. While the overhead of `cilk_spawn` is relatively small, performance will suffer if you spawn very small amounts of work.

OPTIMIZE THE SERIAL PROGRAM FIRST

The first step is to assure that the C++ serial program has good performance and that normal optimization methods, including compiler optimization, have already been used.

As one simple, and limited, illustration of the importance of serial program optimization, consider the `matrix_multiply` example, which organizes the loop with the intent of minimizing cache line misses. The resulting code is:

```
    cilk_for(unsigned int i = 0; i < n; ++i) {
        for (unsigned int k = 0; k < n; ++k) {
            for (unsigned int j = 0; j < n; ++j) {
                A[i*n + j] += B[i*n + k] * C[k*n + j];
            }
        }
    }
```

In multiple performance tests, this organization has shown a significant performance advantage compared to the same program with the two inner loops (the k and j loops) interchanged. This performance difference shows up in both the serial and Cilk++ parallel programs. The `matrix` example has a similar loop structure. Be aware, however, that such performance improvements cannot be assured on all systems as there are numerous architectural factors that can affect performance.

The following article: ***Making Your Cache Go Further in These Troubled Times*** (http://software.intel.com/en-us/articles/Making-Your-Cache-Go-Further-in-These-Troubled-Times) also discusses this topic.

The `wc-cilk` example in another instance showing the advantages of code optimization; using an inline function to detect alphanumeric characters and removing redundant function calls produced significant serial program gains which are reflected in the Cilk++ parallel program.

TIMING PROGRAMS AND PROGRAM SEGMENTS

You must measure performance to find and understand bottlenecks. Even small changes in a program can lead to large and sometimes surprising performance differences. The only reliable way to tune performance is to measure frequently, and preferably on a mix of different systems. The Cilk++ examples use the cilkview object to measure performance. Use any tool or technique at your disposal, but only true measurements will determine if your optimizations are effective.

Performance measurements can be misleading, however, so it is important to take a few precautions and be aware of potential performance anomalies. Most of these precautions are straight-forward but may be overlooked in practice.

- Other running applications can affect performance measurements. Even an idle version of a program such as Microsoft Word can consume processor time and distort measurements.
- If you are measuring time between points in the program, be careful not to measure elapsed time between two points if other strands could be running in parallel with the function containing the starting point.
- ***Dynamic frequency scaling*** (http://en.wikipedia.org/wiki/Dynamic_frequency_scaling) on multicore laptops and other systems can produce unexpected results, especially when you increase worker count to use additional cores. As you add workers and activate cores, the system might adjust clock rates to reduce power consumption and therefore reduce overall performance.

COMMON PERFORMANCE PITFALLS

If a program has sufficient parallelism and burdened parallelism but still doesn't achieve good speedup, the performance could be affected by other factors. Here are a few common factors, some of which are discussed elsewhere.

- **cilk_for Grainsize Setting** (Page 43). If the grain size is too large, the program's logical parallelism decreases. If the grain size is too small, overhead associated with each spawn could compromise the parallelism benefits. The Cilk++ compiler and runtime system use a default formula to calculate the grain size. The default works well under most circumstances. If your Cilk++ program uses `cilk_for`, experiment with different grain sizes to tune performance.
- **Lock contention** (Page 94). Locks generally reduce program parallelism and therefore affect performance. `cilkview` does not account for lock contention when calculating parallelism. Lock usage can be analyzed using other performance and profiling tools.
- **Cache efficiency and memory bandwidth** (Page 119). See the next section.
- **False sharing** (Page 120). See the section later in this chapter.
- **Atomic operations** (Page 122). Atomic operations, provided by compiler intrinsics, lock cache lines. Therefore, these operations can impact performance the same way that lock contention does. Also, since an entire cache line is locked, there can be false sharing. `cilkview` report counts the number of atomic operations; this count can vary between Windows* and Linux* versions of the same program.

CACHE EFFICIENCY AND MEMORY BANDWIDTH

Good cache efficiency is important for serial programs, and it becomes even more important for parallel programs running on multicore machines. The cores contend for bus bandwidth, limiting how quickly data that can be transferred between memory and the processors. Therefore, consider cache efficiency and data and spatial locality when designing and implementing parallel programs. For example code that considers these issues, see the following article: ***Making Your Cache Go Further in These Troubled Times*** http://software.intel.com/en-us/articles/Making-Your-Cache-Go-Further-in-These-Troubled-Times or the `matrix` and `matrix_multiply` examples cited in the "***Optimize the Serial Program First*** (Page 117)" section.

A simple way to identify bandwidth problems is to run multiple copies of the serial program simultaneously—one for each core on your system. If the average running time of the serial programs is much larger than the time of running just one copy of the program, it is likely that the program is saturating system bandwidth. The cause could be memory bandwidth limits or, perhaps, disk or network I/O bandwidth limits.

These bandwidth performance effects are frequently system-specific. For example, when running the matrix example on a specific system with two cores (call it "S2C"), the "iterative parallel" version was considerably slower than the "iterative sequential" version (4.431 seconds compared to 1.435 seconds). On other systems, however, the iterative parallel version showed nearly linear speedup when tested with as many as 16 cores and workers. Here are the results on S2C:

```
1) Naive, Iterative Algorithm. Sequential and Parallel.
Running Iterative Sequential version...
  Iterative Sequential version took 1.435 seconds.
Running Iterative Parallel version...
  Iterative Parallel version took   4.431 seconds.
  Parallel Speedup: 0.323855
```

There are multiple, often complex and unpredictable, reasons that memory bandwidth is better on one system than another (e.g.; DRAM speed, number of memory channels, cache and page table architecture, number of CPUs on a single die, etc.). Be aware that such effects are possible and may cause unexpected and inconsistent performance results. This situation is inherent to parallel programs and is not unique to Cilk++ programs.

FALSE SHARING

False sharing (Page 123) is a common problem in shared memory parallel processing. It occurs when two or more cores hold a copy of the same memory cache line.

If one core writes, the cache line holding the memory line is invalidated on other cores. This means that even though another core may not be using that data (reading or writing), it might be using another element of data on the same cache line. The second core will need to reload the line before it can access its own data again.

Thus, the cache hardware ensures data coherency, but at a potentially high performance cost if false sharing is frequent. A good technique to identify false sharing problems is to catch unexpected sharp increases in last-level cache misses using hardware counters or other performance tools.

As a simple example, consider a spawned function with a `cilk_for` loop that increments array values. The array is `volatile` to force the compiler to generate store instructions rather than hold values in registers or optimize the loop.

```
volatile int x[32];

void f(volatile int *p)
{
    for (int i = 0; i < 100000000; i++)
    {
        ++p[0];
        ++p[16];
    }
}

int cilk_main()
{
    cilk_spawn f(&x[0]);
    cilk_spawn f(&x[1]);
    cilk_spawn f(&x[2]);
    cilk_spawn f(&x[3]);
    cilk_sync;
    return 0;
}
```

The `a[]` elements are four bytes wide, and a 64-byte cache line (normal on x86 systems) would hold 16 elements. There are no data races, and the results will be correct when the loop completes. `cilkview` shows significant parallelism. However, cache line contention as the individual strands update adjacent array elements can degrade performance, sometimes significantly. For example, one test on a 16-core system showed one worker performing about 40 times faster than 16 workers, although results can vary significantly on different systems.

Glossary of Terms

A

About the Glossary

The Glossary is an alphabetical list of important terms used in this programmer's guide and gives brief explanations and definitions.

Terms in ***bold italic*** occur elsewhere in the glossary.

atomic

Indivisible. An ***instruction*** sequence executed by a ***strand*** is atomic if it appears at any moment to any other strand as if either no instructions in the sequence have been executed or all instructions in the sequence have been executed.

C

chip multiprocessor

A general-purpose ***multiprocessor*** implemented as a single ***multicore*** chip.

Cilk

A simple set of extensions to the C programming language that allow a programmer to express concurrency easily. For more information, see MIT's ***Cilk website*** (http://supertech.csail.mit.edu/cilk/).

Cilk Arts, Inc.

An MIT-spinoff founded in 2006 to commercialize the ***Cilk*** technology. The assets of Cilk Arts, Inc. were acquired by Intel® Corporation in July, 2009.

cilk_for

A keyword in the Cilk++ language that indicates a `for` loop whose iterations can be executed independently in parallel.

cilk_spawn

A keyword in the Cilk++ language that indicates that the named subroutine can execute independently and in parallel with the caller.

cilk_sync

A keyword in the Cilk++ language that indicates that all functions spawned within the current function must complete before statements following the `cilk_sync` can be executed.

Cilk++

A simple set of extensions to the C++ programming language that allow a programmer to express concurrency easily. The Cilk++ language is based on concepts developed for the Cilk programming language at MIT.

Cilkscreen

The ***cilkscreen race detector*** is a tool provided in the Intel® Cilk++ SDK for finding ***race condition*** defects in ***Cilk++*** code.

commutative operation

An operation (`op`), over a type (`T`), is commutative if `a op b = b op` **a** for any two objects, `a` and `b`, of type `T`. Integer addition and set union are commutative, but string concatenation is not.

concurrent agent

A ***processor***, ***process***, ***thread***, ***strand***, or other entity that executes a program ***instruction*** sequence in a computing environment containing other such entities.

core

A single ***processor*** unit of a ***multicore*** chip. The terms "processor" and "***CPU***" are often used in place of "core", although industry usage varies.

Archaic: A solid-state memory made of magnetized toroidal memory elements.

CPU

"Central Processing Unit"; we use this term as a synonym for "***core***", or a single ***processor*** of a ***multicore*** chip.

critical section

The code executed by a ***strand*** while holding a ***lock***.

critical-path length

See ***span***.

D

data race

A ***race condition*** that occurs when two or more parallel strands, holding no ***lock*** in common, access the same memory location and at least one of the strands performs a write. Compare with ***determinacy race***.

deadlock

A situation when two or more ***strand*** instances are each waiting for another to release a resource, and the "waiting-for" relation forms a cycle so that none can ever proceed.

determinacy race

A ***race condition*** that occurs when two parallel strands access the same memory location and at least one ***strand*** performs a write.

determinism

The property of a program when it behaves identically from run to run when executed on the same inputs. Deterministic programs are usually easier to debug.

distributed memory

Computer storage that is partitioned among several ***processors***. A distributed-memory ***multiprocessor*** is a computer in which processors must send messages to remote processors to access data in remote processor memory. Contrast with ***shared memory***.

E

execution time

How long a program takes to execute on a given computer system. Also called ***running time***.

F

fake lock

A construct that `cilkscreen` treats as a ***lock*** but which behaves like a no-op during actual running of the program. A fake lock can be used to suppress the reporting of an intentional ***race condition***.

false sharing

The situation that occurs when two strands access different memory locations residing on the same cache block, thereby contending for the cache block.For more information, see the ***Wikipedia entry*** http://en.wikipedia.org/wiki/False_sharing .

G

global variable

A variable that is bound outside of all local scopes. See also ***nonlocal variable***.

H

hyperobject

A linguistic construct supported by the Cilk++ runtime that allows many strands to coordinate in updating a shared variable or data structure independently by providing each strand a different ***view*** of the hyperobject to different strands at the same time. The ***reducer*** is the only hyperobject currently provided by the Intel® Cilk++ SDK.

I

instruction

A single operation executed by a ***processor***.

K

knot

A point at which the end of one ***strand*** meets the end of another. If a knot has one incoming strand and one outgoing strand, it is a *serial knot*. If it has one incoming strand and two outgoing strands, it is a *spawn knot*. If it has multiple incoming strands and one outgoing strand, it is a *sync knot*. A Cilk++ program does not produce serial knots or knots with both multiple incoming and multiple outgoing strands.

L

linear speedup

Speedup proportional to the ***processor*** count. See also ***perfect linear speedup***.

lock

A synchronization mechanism for providing ***atomic*** operation by limiting concurrent access to a resource. Important operations on locks include acquire (lock) and release (unlock). Many locks are implemented as a ***mutex***, whereby only one ***strand*** can hold the lock at any time.

lock contention

The situation wherein multiple strands vie for the same ***lock***.

M

multicore

A semiconductor chip containing more than one ***processor core***.

multiprocessor

A computer containing multiple general-purpose ***processors***.

mutex

A "mutually exclusive" ***lock*** that only one ***strand*** can acquire at a time, thereby ensuring that only one strand executes the ***critical section*** protected by the mutex at a time. Windows* OS supports several mutually exclusive locks, including the `CRITICAL_SECTION`. Linux* OS supports Pthreads `pthread_mutex_t` objects.

N

nondeterminism

The property of a program when it behaves differently from run to run when executed on exactly the same inputs. Nondeterministic programs are usually hard to debug.

nonlocal variable

A program variable that is bound outside of the scope of the function, method, or class in which it is used. In Cilk++ programs, we also use this term to refer to variables with a scope outside a `cilk_for` loop.

P

parallel loop

A `for` loop all of whose iterations can be run independently in parallel. The ***cilk_for*** keyword designates a parallel loop.

parallelism

The ratio of ***work*** to ***span***, which is the largest ***speedup*** an application could possibly attain when run on an infinite number of processors.

perfect linear speedup

Speedup equal to the ***processor*** count. See also ***linear speedup***.

process

A self-contained ***concurrent agent*** that by default executes a serial chain of instructions. More than one ***thread*** may run within a process, but a process does not usually share memory with other processes. Scheduling of processes is typically managed by the operating system.

processor

A processor implements the logic to execute program instructions sequentially; we use the term "***core***" as a synonym. This document does not use the term "processor" to refer to multiple processing units on the same or multiple chips, although other documents may use the term that way.

R

race condition

A source of ***nondeterminism*** whereby the result of a concurrent computation depends on the timing or relative order of the execution of instructions in each individual ***strand***.

receiver

A variable to receive the result of the function call.

reducer

A ***hyperobject*** with a defined (usually associative) `reduce()` binary operator which the ***Cilk++*** runtime system uses to combine the each ***view*** of each separate ***strand***.

A reducer must have two methods:

- A default constructor which initializes the reducer to its identity value
- A `reduce()` method which merges the value of right reducer into the left (this) reducer.

response time

The time it takes to execute a computation from the time a human user provides an input to the time the user gets the result.

running time

How long a program takes to execute on a given computer system. Also called ***execution time***.

S

scale down

The ability of a parallel application to run efficiently on one or a small number of processors.

scale out

The ability to run multiple copies of an application efficiently on a large number of processors.

scale up

The ability of a parallel application to run efficiently on a large number of ***processors***. See also ***linear speedup***.

sequential consistency

The memory model for concurrency wherein the effect of ***concurrent agents*** is as if their operations on ***shared memory*** were interleaved in a global order consistent with the orders in which each agent executed them. This model was advanced in 1976 by ***Leslie Lamport*** http://research.microsoft.com/en-us/um/people/lamport/

serial execution

Execution of the ***serialization*** of a Cilk++ program.

serial semantics

The behavior of a Cilk++ program when executed as the ***serialization*** of the program. See the following article: "***Four Reasons Why Parallel Programs Should Have Serial Semantics***".

serialization

The C++ program that results from stubbing out the keywords of a ***Cilk++*** program, where ***cilk_spawn*** and ***cilk_sync*** are elided and ***cilk_for*** is replaced with an ordinary `for`. The serialization can be used for debugging and, in the case of a converted C++ program, will behave exactly as the original C++ program. The term "*serial elision*" is used in some of the literature. Also, see "***serial semantics***".

shared memory

Computer storage that is shared among several processors. A shared-memory ***multiprocessor*** is a computer in which each ***processor*** can directly address any memory location. Contrast with ***distributed memory***.

span

The theoretically fastest execution time for a parallel program when run on an infinite number of ***processors***, discounting overheads for communication and scheduling. Often denoted by T_∞ in the literature, and sometimes called ***critical-path length***.

spawn

To call a function without waiting for it to return, as in a normal call. The caller can continue to execute in parallel with the called function. See also ***cilk_spawn***.

speedup

How many times faster a program is when run in parallel than when run on one ***processor***. Speedup can be computed by dividing the running time T_P of the program on P processors by its running time T_1 on one processor.

strand

A ***concurrent agent*** consisting of a serial chain of instructions without any parallel control (such as a ***spawn***, ***sync***, return from a ***spawn***, etc.).

sync

To wait for a set of *spawned* functions to return before proceeding. The current function is dependent upon the spawned functions and cannot proceed in parallel with them. See also ***cilk_sync***.

T

thread

A *concurrent agent* consisting of a serial ***instruction*** chain. Threads in the same job share memory. Scheduling of threads is typically managed by the operating system.

throughput

A number of operations performed per unit time.

V

view

The state of a ***hyperobject*** as seen by a given ***strand***.

W

work

The running time of a program when run on one ***processor***, sometimes denoted by T_1.

work stealing

A scheduling strategy where processors post parallel work locally and, when a ***processor*** runs out of local work, it steals work from another processor. Work-stealing schedulers are notable for their efficiency, because they incur no communication or synchronization overhead when there is ample ***parallelism***. The ***Cilk++*** runtime system employs a work-stealing scheduler.

worker

A ***thread*** that, together with other workers, implements the ***Cilk++*** runtime system's ***work stealing*** scheduler.

Index