CILK++ PROGRAMMER'S GUIDE

Cilk++ for Linux – Version 1.0.3

Cilk++ for Windows – Version 1.0.3

Revision 58, March 16, 2009

CONTENTS

Chapter 1

CILK++ PROGRAMMER'S GUIDE INTRODUCTION

Revision 58 (March 16, 2009). Click ***here*** http://web.cilk.com/_packages/documentation/ to get the most recent "Cilk++ Programmer's Guide" revision.

This Cilk++ Programmer's Guide covers two products:

- Cilk++ for Linux — Version 1.0.3. The Cilk++ compiler is based on the ***GNU Compiler Collection*** (http://gcc.gnu.org/)(GCC version 4.2.4)
- Cilk++ for Windows — Version 1.0.3

The two products are very similar, and differences are clearly marked in the text.

TARGET AUDIENCE

The programmer's guide is designed for application developers who will use Cilk++ to improve performance by adding parallelism to new and existing C++ applications. Parallelism enables applications to exploit multiple processor ***cores*** (see "core" Page 152) or ***CPUs*** ("CPU" Page 152) to increase performance. Using Cilk++, you can gain significant performance enhancement for many applications running on Linux and Windows systems ranging from laptops to enterprise servers.

WHAT THIS PROGRAMMER'S GUIDE COVERS

- Installation, along with hardware and software requirements
- Getting started with the Cilk++ language and parallelism
- Compiling, running, and debugging Cilk++ programs
- Example programs which illustrate Cilk++ principles and techniques
- The Cilk++ language, which extends C++ to provide parallelism
- The Cilk++ runtime system and libraries
- Cilk++ reducers, which remove race conditions resulting from shared variables
- Race conditions and how to avoid them
- The Cilkscreen™ Race Detector and Parallel Performance Analyzer tools
- How to mix C++ and Cilk++ in large applications
- Strategies and techniques to identify and exploit application parallelism
- Target platform (Linux or Windows) specific issues as well as portability requirements
- Limitations and restrictions

- Background reference material for those interested in Cilk++ history and parallelism theory.

ASSUMED READER KNOWLEDGE AND BACKGROUND

Basic C++ language knowledge is assumed. In-depth knowledge of C++ templates will be helpful in some places but not required. Likewise, knowledge of and previous experience with parallelism and multithreaded programming may be helpful but not required.

FEEDBACK, SUPPORT, AND BUG REPORTING

Cilk Arts is committed to continuous product improvement and to helping our users get the most out of Cilk++. If you have *feedback*, *questions*, or *problems* with this Programmer's Guide or Cilk++, or would like to *report a bug*, please contact **support[at]cilk[dot]com**.

Please provide sufficient information to help us respond promptly. Useful information can include:

- Complete problem description
- Source code where appropriate
- Error or warning messages

RELEASE NOTES

- The Release Notes are included in the release download. The Release Notes list bug fixes, new features, documentation updates, new examples, limitations, and other changes.
- The release notes are also available online:
  1. ***Cilk++ for Linux Release Notes*** http://web.cilk.com/_packages/linuxreleasenotes.html
  2. ***Cilk++ for Windows Release Notes*** http://web.cilk.com/_packages/windowsreleasenotes.html

ADDITIONAL RESOURCES AND INFORMATION

There is a wealth of additional and supplementary information available about Cilk++ on the ***Cilk Arts web site*** (http://www.cilk.com/) and elsewhere. Here are some recommended references if you want to explore Cilk++ in greater depth.

- ***"How to Survive the Multicore Revolution (or at Least Survive the Hype)"*** (http://www.cilk.com/multicore-e-book/). This free Cilk Arts e-Book provides an overview of parallelism and is a excellent introduction to Cilk++.
- The "Cilk++ Technology" pull down menu on ***Cilk Arts web site*** (http://www.cilk.com/) contains discussions and video presentation links covering many Cilk++ features.

- ***Cilk Arts blog*** (http://www.cilk.com/resource-library/resources-for-multicoders/) with solutions and sample code for numerous topics such as:
    3. ***Global variables*** (http://www.cilk.com/multicore-blog/bid/5672/Global-Variable-Reconsidered)
    4. ***Cilk++ matrix multiplication*** (http://www.cilk.com/multicore-blog/bid/6248/A-Tale-of-Two-Algorithms-Multithreading-Matrix-Multiplication)
    5. ***Cilk++ performance and cache management*** http://www.cilk.com/multicore-blog/bid/6934/Making-Your-Cache-Go-Further-in-These-Troubled-Times
    6. ***Cilk++ Sets World Record for Crypto Hash Function Throughput*** http://www.cilk.com/multicore-blog/bid/7412/Cilk-Sets-World-Record-for-Crypto-Hash-Function-Throughput
- The ***Cilk Implementation Project site*** (http://supertech.csail.mit.edu/cilkImp.html) is a gateway to the MIT Cilk project which developed numerous ideas that influenced Cilk++'s development. 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 (Cilk Arts Chief Scientist), Charels E. Leiserson (Cilk Arts Founder), and Keith H. Randall, won the ***Most Influential 1998 PLDI Paper award*** (http://www.cilk.com/multicore-blog/bid/5607/Cilk-Wins-Most-Influential-PLDI-Paper-Award) at the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. This paper also provides background information about technology that has influenced Cilk++ development.
- ***Presentations and Technical Talks*** (http://www.cilk.com/resource-library/presentations-technical-talks/) has links covering a variety of business and technical subjects.

Chapter 2

LINUX INSTALLATION

This chapter is for ***Cilk++ for Linux*** programmers only.

The hardware and software requirements for installing Cilk++ for Linux are listed here. The next section gives the installation steps.

HARDWARE PREREQUISITES - CILK++ FOR LINUX

You can build and test Cilk++ programs on single and multiple ***core*** (Page 152) machines as well as on virtual machines; multiple cores are recommended for effective testing and performance measurement.

Cilk++ is designed for ***shared memory*** (Page 155) multiprocessor systems. Parallel constructs in Cilk++ programs can use as many cores as are available.

Cilk++ for Linux has passed tests on systems with as many as 32 cores. Previous Cilk versions have scaled will to systems with more than 1,000 cores.

The processors must be Intel x86 or x86-64 architecture (Pentium 3 or later), in a shared memory system with one or more processors and any number of multicore and multiple socket configurations.

SOFTWARE PREREQUISITES - CILK++ FOR LINUX

This Cilk++ for Linux version requires a Linux distribution built for systems with Intel x86 or x86-64 architecture processors (32 or 64 bit, respectively). **Note:** We have successfully tested Cilk++ on several Linux versions and distributions, including Fedora and Debian.

The Cilk++ compiler can run on both 32-bit and 64-bit Linux systems, and both can produce either 32-bit and 64-bit executables, based on command line options. There are two "Cilk++ for Linux" versions, or packages (32-bit and 64-bit).

- We recommend that you install the 64-bit package, not the 32-bit package, on 64-bit systems.
- If you do install the 32-bit package on a 64-bit system (that is, in a shared development environment that supports both 32 and 64-bit OSs), then you should explicitly use "`-m32`" or "`-m64`" to ***compile programs*** (see "Building from the Linux Command Line" Page 29).
- The 64-bit package can only be installed on 64-bit systems.

Cilk++ is based on GCC version 4.2.4; the user must provide a compatible linker and assembler.

INSTALLING CILK++ FOR LINUX

Cilk++ is distributed as a compressed tar file package. The file name contains the build number and target architecture. For example:

```
cilk_6605-x86_64.release.tar.gz
```

is the package representing release build 6252 for the x86-64 architecture. In this section, the package will simply be called `cilk.tar.gz`.

If you wish to install into a shared, system-wide location, you require root privileges. Login as **root** using the `su` command and follow these steps in order:

1. The default Cilk++ installation location (*install_directory*) is the `/usr/local/` directory.
2. Copy `cilk.tar.gz` to a convenient location (the *targz_directory*), such as the home directory.
3. Change your working directory to the *install_directory* with the Linux command: `cd /usr/local`
4. Install Cilk++ in the *install_directory*, `/usr/local/`, with the command:
   `tar -xzf` *`targz_directory`*`/cilk.tar.gz`

This completes the installation. Logout as root.

To install as a user in your home directory, the process is nearly the same; just the *install_directory* is different, and you do not need root privileges.

5. Copy `cilk.tar.gz` to a convenient location (the *targz_directory*), such as the home directory.
6. Change the working directory to the *install_directory* location where Cilk++ will be installed.
7. Install Cilk++ in the *install_directory* with the command:
   `tar -xzf` *`targz_directory`*`/cilk.tar.gz`

This completes the installation; Cilk++ has been installed in the *install_directory*`/cilk` directory.

Once installation is complete, set the `PATH` environment variable with the command:

```
PATH=install_directory/cilk/bin:$PATH
```

The Cilk++ programming tools and libraries will be installed in the *install_directory*`/cilk/bin` directory. The tools include the `cilk++` compiler as well as Cilk++ versions of the GCC tools: `c++`, `cpp`, `g++`, `gcc`, and `gdb`.

The installation process installs the examples in the directory:

```
install_directory/cilk/examples
```

Furthermore, this process installs the Cilk++ ***Runtime System and Libraries*** (Page 61), which are required to run Cilk++ programs. The Cilk++ Runtime Library may be used by the developer for development and testing. In order to deploy a Cilk++ program, you must redistribute the Runtime under the terms of the ***Cilk Arts Public License (CAPL)*** (Page 137) or under the terms of a Commercial License purchased from Cilk Arts.

If you install the Open Source Evaluation version of Cilk++, some of the tools are available under a 30-day trial license. Contact Cilk Arts for information about purchasing copies of these tools.

Chapter 3

WINDOWS INSTALLATION

This chapter is for ***Cilk++ for Windows*** programmers only.

The hardware and software requirements for installing Cilk++ for Windows are listed here. The following sections give the installation and uninstall steps.

HARDWARE PREREQUISITES - CILK++ FOR WINDOWS

You can build and test Cilk++ programs on single and multiple ***core*** (Page 152) machines and on a virtual machine. Cilk++ is designed for ***shared memory*** (Page 155) multiprocessor systems. Multiple cores are recommended for effective testing and performance measurement.

Parallel constructs in Cilk++ programs can use as many cores as are available.

Cilk++ for Windows has run tests successfully on systems with as many as 16 cores. Previous versions of Cilk software have scaled well to systems with more than 1,000 cores.

The processors must be Intel x86 architecture (Pentium 3 or later), in a shared memory system with one or more processors and any number of multicore and multiple socket configurations.

Memory, disk, clock speed, and other requirements are the same as for ***Visual Studio 2005*** http://msdn.microsoft.com/en-us/library/4c26cc39.aspx or for ***Visual Studio 2008*** http://msdn.microsoft.com/en-us/library/4c26cc39.aspx.

SOFTWARE PREREQUISITES - CILK++ FOR WINDOWS

This version of Cilk++ requires Microsoft Visual Studio 2008 (including Visual Studio 2008 Express) or Microsoft Visual Studio 2005, with Service Pack 1 installed. Install "Visual Studio 2005 with Service Pack 1 Update for Windows Vista" when using VS 2005 with Vista. Cilk++ does NOT work with Visual Studio 2005 Express Edition.

Visual Studio 2008 Express does not allow add-ins, so you will have reduced flexibility as described in the "***Visual Studio with Cilk++ for Windows*** (Page 35)" section.

Cilk++ has been tested on multiple versions of Microsoft Windows, including Microsoft Windows XP with Service Pack (SP) 2 and SP3, Microsoft Windows Vista, and Windows Server 2003 Standard Edition with SP1.

Cilk++ for Windows creates a 32-bit Cilk++ executable for x86 processors.

INSTALLING CILK++ FOR WINDOWS

Before installing Cilk++, verify the following requirements:

- You must have Administrator privileges to install Cilk++
- Uninstall any earlier Cilk++ version. Also exit any command shells from which you expect to execute Cilk++ programs or perform builds, since reinstallation may change environment settings.
- Confirm that the target installation directory (normally "`Program Files\Cilk Arts`") was totally removed by the uninstall; if you changed any files in that directory, these files might not be deleted.
- Exit any open instances of Microsoft Visual Studio before installing or uninstalling Cilk++.

Cilk++ installation is a two-step process to install the main Cilk++ system and the examples.

Cilk++ is distributed as a Windows Installer package: `cilk.msi`. Install Cilk++ by double-clicking on the `cilk.msi` file. Installation takes several minutes.

By default, the Cilk++ programming tools and libraries will be installed in the

```
Program Files\Cilk Arts\Cilk++
```

directory; you can change the installation directory at installation time. Additional files will be installed in the Visual Studio directories to provide Cilk++ integration into the Visual Studio development environment.

Among other things, this process installs the Cilk++ ***Runtime System and Libraries*** (Page 61) DLL, `cilk10.dll`, which is required to run Cilk++ programs. The Cilk++ Runtime Library may be used by the developer for development and testing. In order to deploy a Cilk++ for Windows program, you will need a runtime license from Cilk Arts.

The evaluation version of Cilk++ may be used for 30 days under the free trial license. Contact Cilk Arts for information about purchasing copies of these tools.

Install the examples by running `cilkexamples.msi` in the

```
Program Files\Cilk Arts\Cilk++
```

directory. The default examples installation location, which you can override, is:

```
%USERPROFILE%\Documents\Cilk++ Examples
```

A Note for Windows Vista Users: If Windows "User Account Control" (UAC) is turned on (the normal setting), you will be prompted for your password as Cilk++ for Windows is installed. UAC is a Vista feature designed to prevent unauthorized changes to your computer; in particular, Cilk++ for Windows is installed into the "Program Files" directory. You can disable UAC though the Control Panel's "Security Center". Consult your Windows Vista documentation or visit Microsoft's "***User Account Control Step-by-Step Guide*** http://technet.microsoft.com/en-us/library/cc709691.aspx". This information also applies to Windows Server 2008.

UNINSTALLING CILK++ FOR WINDOWS

You can uninstall both Cilk++ and the Cilk++ Examples using the Windows Control Panel.

- Windows XP: **Add or Remove Programs** in the Windows Control Panel.
- Windows Vista: Use either **Programs Uninstall a program** or **Programs -> Programs and Features** in the Control Panel.

Modified files will not be removed when you uninstall. You should consider deleting the directory where the examples were installed after uninstalling the Cilk++ examples.

Chapter 4

GETTING STARTED

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

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

The sub-problems may be implemented in separate functions or in iterations of a loop. Cilk++ provides the `cilk_spawn` keyword to identify functions that can run in parallel and the `cilk_for` keyword to identify loops that can run in parallel.

Note that with Cilk++, you do not need to specify the number of processors or decide which code runs on which processor. The Cilk++ runtime system performs all the scheduling necessary to run programs efficiently on the available processors.

This chapter introduces Cilk++ by converting a simple C++ application to Cilk++, building and running the Cilk++ application, and comparing performance as a function of ***worker*** (Page 156) count.

The first step, even before the introduction to the conversion process and Cilk++ language features, is to sample Cilk++'s power trying one of the ***Cilk++ Examples*** (Page 39).

GETTING STARTED: RUNNING CILK++ EXAMPLES

This section shows how to build, run, and test one of the ***Cilk++ Examples*** (Page 39) to see Cilk++'s capabilities and ease of use. Following that, we will introduce Cilk++ language and library features by converting a C++ application to Cilk++.

The examples are installed in individual folders, as described in the ***Linux Installation*** (Page 11) and ***Windows Installation*** (Page 15) chapters.

SELECTING AN EXAMPLE

`qsort` is the best "getting started" example, since it is the basis for the Cilk++ conversion discussion.

Nearly any other example will also illustrate Cilk++'s power on a ***multicore*** (Page 153) system. However:

- Do not use `qsort-cpp`, as it is a C++ program.
- `QuickDemo` is **Windows only**, but it is a good choice for Windows users. A GUI shows the computation progress and compares serial and parallel performance.

- Do not use `qsort-dll` as it is **Windows only** and also does not allow you to set the worker count.
- Do not use `qsort-race` as it contains an intentional data race bug.

BUILDING AN EXAMPLE

The full details are in the "***Building, Running, and Debugging*** (Page 29)" chapter, with separate sections for Linux and Windows. Assume that:

- `qsort` is the selected example
- Cilk++ is properly installed on your multicore system
- The `PATH` environment variable is set properly so as to access the Cilk++ compiler.

Linux Only: Change to the example directory (`.../examples/qsort`) and execute `make`. The executable, `qsort`, will be built in the example directory.

Windows Only:

- Visual Studio 2005 and 2008 users can open the solution (such as `qsort.sln`) and build the release version. The executable is `...\examples\qsort\release\qsort.exe`.
- From the command line, build with the command:

  ```
  cilkpp qsort.cilk
  ```

  The executable is `...\examples\qsort\qsort.exe`.

RUNNING AN EXAMPLE

Forward slash and back slash: When illustrating use in the Windows environment, this document uses back slash ("\") as the pathname separator. Forward slash ("/") is used in the Linux environment. When the environment could be either Linux or Windows, we use the Linux forward slash.

The examples allow worker count specification using the `-cilk_set_worker_count` option, and they report the execution time. Here are some results on an 8-core system, where the speedup is limited by the application's parallelism and the core count.

```
examples/qsort$ qsort 10000000 -cilk_set_worker_count=1
Sorting 10000000 integers
2.909 seconds
Sort succeeded.

examples/qsort$ qsort 10000000 -cilk_set_worker_count=2
Sorting 10000000 integers
1.468 seconds
Sort succeeded.

examples/qsort$ qsort 10000000 -cilk_set_worker_count=4
Sorting 10000000 integers
```

```
0.798 seconds
Sort succeeded.

examples/qsort$ qsort 10000000 -cilk_set_worker_count=8
Sorting 10000000 integers
0.438 seconds
Sort succeeded.

examples/qsort$ qsort 10000000 -cilk_set_worker_count=16
Sorting 10000000 integers
0.548 seconds
Sort succeeded.
```

CILK++ CONVERSION OVERVIEW

Here is an overview of the sequence of steps to create a parallel program using Cilk++. Complete details are in subsequent sections.

1. 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.
2. 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. As an example of this process, see the Cilk Arts blog, "***Finding Performance Bottlenecks & Data Races*** http://www.cilk.com/multicore-blog/bid/7454/Finding-Performance-Bottlenecks-Data-Races".
3. Rename the Cilk++ source files, replacing the `.cpp` extension with `.cilk`.
   - **Windows Specific:** When using Visual Studio, use the "Convert to Cilk" context menu.
4. Use the three Cilk++ keywords to mark program parts 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.
5. Build the program:
   - **Windows Specific:** Use either the `cilkpp` command-line tool or Visual Studio.
   - **Linux Specific:** Use the `cilk++` compiler command.
6. Run the program. If there are no ***race conditions*** (Page 67), the parallel program will produce the same result as the serial program.
7. 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 75) to identify possible race conditions introduced by parallel operations.

8. ***Correct any race conditions*** (see "Data Race Correction Methods" Page 68) using Cilk++ ***Reducers*** (Page 85), locks, or recoding to resolve conflicts.
9. Note that a traditional debugger can debug the *serialization* (Page 155) of a parallel program, which you can create easily with Cilk++.

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

STARTING WITH A SERIAL PROGRAM

We'll demonstrate how to use Cilk++ by parallelizing a simple implementation of ***Quicksort*** (http://en.wikipedia.org/wiki/Quicksort). The source files are in the Cilk++ installation `examples` directory.

- The original serial program, `qsort.cpp`, is in the `qsort-cpp` directory.
- The completed Cilk++ program, `qsort.cilk`, is in the `qsort` directory.

The next chapter describes how to build the Cilk++ code from the ***command line*** (see "Building, Running, and Debugging" Page 29) and ***Windows Visual Studio*** (see "Visual Studio with Cilk++ for Windows" Page 35). This section describes the code modifications required to convert the C++ code to Cilk++.

The sample implementation below uses the STL `partition` algorithm and the STL `less` and `bind2nd` adaptors. The STL details are not important; the key point is that we will convert an existing C++ program to Cilk++.

The include file, `example_util_gettime.h`, defines a millisecond timer function, `example_get_time()`; it is not specific to Cilk++. All the examples use this function.

Notice that the sorting 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>
10  #include "../include/example_util_gettime.h"
11  #include <iostream>
12  #include <iterator>
13  #include <functional>
14
15  // Sort the range between begin and end.
16  // end is one past the final element in the range.
17  // Quick Sort algorithm, recursive divide-and-conquer.
18  // NOT the same as the Standard C Library qsort().
19  // This is pure C++ code before Cilk++ conversion.
20
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));
27          using std::swap;
28          swap(*end, *middle);    // move pivot to middle
29          sample_qsort(begin, middle);
30          sample_qsort(++middle, ++end); // Exclude pivot
31      }
32  }
33
34  // A simple test harness
35  int qmain(int n)
36  {
37      int *a = new int[n];
38
39      for (int i = 0; i < n; ++i)
40          a[i] = i;
41
42      std::random_shuffle(a, a + n);
43      std::cout << "Sorting " << n << " integers"
                  << std::endl;
44      long t1 = example_get_time();
45      sample_qsort(a, a + n);
46      long t2 = example_get_time();
47      std::cout << (t2 - t1) / 1000.f << " seconds"
                  << std::endl;
48
49      // Confirm that a is sorted and that each element
        //   contains the index.
50      for (int i = 0; i < n-1; ++i) {
51          if ( a[i] >= a[i+1] || a[i] != i ) {
52              std::cout << "Sort failed at location i="
                          << i << " a[i] = "
53                        << a[i] << " a[i+1] = " << a[i+1]
                          << std::endl;
54              delete[] a;
55              return 1;
56          }
57      }
58      std::cout << "Sort succeeded." << std::endl;
59      delete[] a;
60      return 0;
61  }
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  }
```

Note that, on lines 29 and 30, `sample_qsort` calls itself twice recursively. Since the second recursive call does not depend on the results of the first recursive call, we have an opportunity to speed up the call by allowing both calls to run in parallel.

Lines 44 and 46 measure the elapsed time (in milliseconds) required to perform the sort, using the `example_get_time()` function, which returns a 32-bit millisecond timer and is defined in:

`../include/example_util_gettime.h.`

CHANGING THE C++ SOURCE TO CILK++

Converting the C++ code to Cilk++ requires several simple steps. The result is a Cilk++ program that does not yet use any Cilk++ keywords; they will be added next.

- Rename the source file by changing the `.cpp` extension to `.cilk`.
  - **Windows Only:** Visual Studio provides a "Convert to Cilk" context menu item for this purpose.
- Add a "`#include <cilk.h>`" statement to the source. `cilk.h` declares all the entry points in the Cilk++ runtime.
- Rename the `main()` function (Line 63) to `cilk_main()`. Cilk++ programs use this entry point to process the ***command line*** (see "Building and Executing" Page 26), which can contain Cilk++ options as well as program options, and to run the program in a Cilk++ context.
  - There are alternatives to `cilk_main()` using the ***Runtime System and Libraries*** (Page 61).
  - `cilk_main()` takes 0, 2, or 3 arguments with the prototypes:

    ```
    int cilk_main ();
    int cilk_main (int argc, char *argv[]);
    int cilk_main (int argc, char *argv[], char *env[]);
    ```
  - **Windows Only:** `cilk_main()` also supports wide characters and consistency with `wmain()` 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.

BUILD THE RELEASE VERSION

Normally, Cilk++ projects should be built with optimized code for best performance. Therefore, build the release version:

- **Windows command line:** `cilkpp /EHSc qsort.cilk`
- **Windows Visual Studio:** Specify the release version build
- **Linux:** `cilk++ qsort.cilk -o qsort -O2`

There are two situations where a debug build (`-g` compiler option for **Linux**, and "Debug" mode for **Windows Visual Studio** projects) is appropriate:

- You want to build and debug the program's ***serialization*** (Page 155), as described in the "***Debugging Cilk++ Programs*** (Page 37)" chapter.
- You want the diagnostic output from the ***Cilkscreen Race Detector*** (Page 75) to report data races. Using a debug build, Cilkscreen can provide more complete and accurate symbolic information.

SPAWN AND SYNC — GETTING STARTED

We are now ready to add the Cilk++ `cilk_spawn` and `cilk_sync` keywords into our `qsort` example. The `cilk_spawn` keyword says that a specific function can be executed in parallel with the code that follows the `cilk_spawn` statement. The `cilk_sync` statement says that the function will 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));
27          using std::swap;
28          swap(*end, *middle);    // move 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` before returning from a Cilk++ function, so the `cilk_sync` at line 32 is redundant, but recommended, 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. On a large input set, this creates the possibility of a very high degree of parallelism.

PERFORMANCE: A NOTE

The Cilk++ scheduler uses ***CPU*** (Page 152) (or ***core*** (Page 152)) resources efficiently while ensuring 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. The space bound is important, as the number of logical strands grows exponentially with the input size.

Divide-and-conquer is an effective parallelization strategy, creating a good mix of large and small sub-problems. The Cilk++ 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. Cilk++ provides the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) to help determine if the chunk size (or granularity) is optimal.

BUILDING AND EXECUTING

With these changes, you can now build the `qsort` Cilk++ code and execute the program. Build and run the release version as described in the "***Getting Started: Running Cilk++ Examples*** (Page 19)" section.

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 and will exploit your system's resources. However, you can use fewer workers to reserve resources for other tasks.

A Performance Note: In some cases, you may wish to "oversubscribe" the workers; that is, assign more workers than the number of cores. This may be appropriate if the program uses locks (see "***Locks Reduce Parallelism*** (Page 71)"); however, oversubscribing will often reduce performance.

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 also accepts `/cilk_set_worker_count`). `cilk_main()` processes this option and removes it from the command line string that is visible to the application.

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
```

Programs that do not use `cilk_main()`, such as `QuickDemo` and `qsort-dll`, cannot use the option. These examples use the ***Runtime System and Libraries*** (Page 61) to invoke Cilk++ code from C++, and the "***Mixing C++ and Cilk++*** (Page 115)" chapter describes more general situations that do not require `cilk_main()`.

Linux Only: You can specify the number of worker threads using an environment variable, `CILK_NPROC`. For example, run the `qsort` example with a command such as:

```
CILK_NPROC=4 ./qsort 1000
```

CILK_FOR LOOPS — GETTING STARTED

Two Cilk++ keywords, `cilk_spawn` and `cilk_sync`, were sufficient for the `qsort` example. The third keyword is `cilk_for`.

To parallelize a loop, one can use the `cilk_for` loop keyword. The following Cilk++ program illustrates a parallelized loop (see Line 34):

```
 1  // A demonstration of a cilk_for loop.
 2
 3  #include <iostream>
 4  #include <cstdlib>
 5  #include <ctime>
 6  #include <windows.h>
 7  #include <cilk.h>
 8
 9  #include "../include/example_util_gettime.h"
10
11  int dowork(int i)
12  {
13      // Waste time:
14      volatile int j = 0;
15      for (j = 0; j < 50000; ++j)
16          ;
17
18      return i;
19  }
20
```

```
21  int cilk_main(int argc, char *argv[])
22  {
23      int n = 100000;
24      if (argc > 1)
25          n = std::atoi(argv[1]);
26
27      int *a = new int[n];
28      for (int j = 0; j < n; ++j)
29          a[j] = j;
30
30      std::cout << "Iterating over " << n << " integers"
                  << std::endl;
32      long t1 = example_get_time();
33
34      cilk_for (int i = 0; i < n; ++i)
35          dowork(a[i]);
36
37      long t2 = example_get_time();
38      std::cout << (t2 - t1) / 1000.f << " seconds"
39                << std::endl;
40      return 0;
41  }
```

The time-wasting function, `dowork()`, simulates work that uses significant CPU resources.

The `cilk_for` keyword on line 34 indicates to the compiler that each invocation of `dowork()` in the loop at line 35 can be performed in parallel, because it works on an independent variable.

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. This divide-and-conquer strategy allows the Cilk++ scheduler to provide significantly better performance compared to an individual spawn with each loop iteration.

Chapter 5

BUILDING, RUNNING, AND DEBUGGING

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

- Linux command line
- Windows command line
- Windows Visual Studio

Build options are described for each environment.

BUILDING FROM THE LINUX COMMAND LINE

This section is for ***Cilk++ for Linux*** programmers only.

- Compile Cilk++ files using the `cilk++` command in `/usr/local/cilk/bin` or *install_directory*`/cilk/bin`. Remember to set the PATH environment variable to include the appropriate `cilk/bin` directory.
- Files compiled with `cilk++` will be considered to contain Cilk++ code whether the suffix is `.cilk`, `.cpp`, or `.c`. Files compiled with the Cilk++ version of the `g++` command in the same directory will be considered to contain Cilk++ code if they have a `.cilk` suffix.
- Programs compiled with Cilk++ code can access reducers and the Cilk++ context, regardless of the filename suffix. When compiling with either the `gcc` or `g++` command (installed Cilk++ versions), the compiler will treat files with `.cpp` and `.c` suffixes normally (as non-Cilk++).
- In general these commands recognize the same set of options as `g++`. Several additional options related to Cilk are also available and described in the next section.
- 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 Cilk++ program's performance.
- To generate a 32-bit Cilk++ executable, use the `-m32` command line option. To generate a 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++ 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
```

The C++ compiler can process most Cilk++ programs without modification to create a ***serialization*** (Page 155). Serialization is especially useful when ***debugging Cilk++ programs*** (Page 37). To compile the serialization, include `cilk_stub.h` before `cilk.h`.

The compiler provides command line options for serialization that do not require source code changes. 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
```

LINUX CILK++ COMMAND LINE OPTIONS

The Linux Cilk++ compiler (invoked either as `cilk++` or `g++`) supports the following `-W` and `-f` options. The Linux (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 makes no use of Cilk++ features 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 and ordinary `for` loops. This warning is enabled by default.

`-Wcilk-promote`

Warn when a C++ function is converted ("promoted") to a ***Cilk++ function*** (see "Cilk++ and C++ Language Linkage" Page 53). 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.)

`-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++ constructs. This is on by default, even if the program is being compiled as C++. Cilk++ constructs 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 155) of a Cilk++ program. See the example at the end of the preceding section.

`-fcilk-check-spawn-queue`

The 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 due to a program bug causing 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++ frame allocation. This is enabled by default when optimizing and should not need to be disabled.

`-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 Cilk Arts build number, as follows:

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

LINUX CILK++ SHARED LIBRARY CREATION

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++.

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 Cilk++ 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
```

BUILDING FROM THE WINDOWS COMMAND LINE

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

Cilk++ modules (files ending in `.cilk`) are compiled using the `cilkpp` command, which is in the `bin` directory under the Cilk++ installation directory. The `cilkpp` program accepts most `cl` options, including those for code optimization (in general, use the same options as used for a C++ program). It will issue a warning for any options that the Cilk++ compiler does not support.

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

WINDOWS CILK++ COMMAND LINE OPTIONS

`cilkpp` supports the following additional 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 155). 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. `cilkpp`, not the program, is being debugged.

`/cilkp version`

This provides the Cilk Arts build number and version.

`/TK`

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

`cilkpp` processes these options when compiling `.cilk` files.

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.

RUNNING WINDOWS CILK++ PROGRAMS

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 easiest to see with the Windows Task Manager's "Performance" tab; all CPUs may appear to be busy, even if only one strand is executing.

Fortunately, the Cilk++ 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.

If the program only occasionally requires parallel resources, use the explicit `cilk::run()` interface, which is part of the ***runtime system and libraries*** (Page 61). When you call `cilk::run()`, the processors will all be active until `cilk::run()` returns. At that point, the processors should sleep until the next call to `cilk::run()`. There is some additional cost to entering and leaving Cilk++ in this manner, but if there are long periods with no parallel work intermixed with long periods of high parallelism, that cost will be insignificant. See the "***Mixing C++ and Cilk++*** (Page 115)" chapter for an extended discussion of this subject.

VISUAL STUDIO WITH CILK++ FOR WINDOWS

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

Cilk++ is not implemented as a new language in Visual Studio. Rather, it is implemented as a variant of C++. This means that Visual Studio has no notion of a Cilk++ project. This section shows how to convert an existing C++ project to a Cilk++ variant, how to set compiler options, and how to run Cilk++ programs.

VISUAL STUDIO AND CILK++ IN AN EXISTING PROJECT

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 Microsoft C/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: there are 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.

Visual Studio 2008 Express Edition does not support add-ins, so these two options are not available on the context menu. `.cilk` files are supported, but you have to copy all the C/C++ settings by hand. If you must use the Express Edition, it is generally simpler to use the `cilkpp` command line program.

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 Cilk++ 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 55).

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 Cilk++ installation. Visual Studio caches UI elements and may not recognize that the Cilk++ 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\Cilk Arts\Cilk++\visualstudio"`
- Run `ResetAddin.bat`

Alternatively, from **Start** - **run**, execute:

```
devenv /resetaddin Cilk.Connect
```

VISUAL STUDIO COMPILER OPTIONS

The ***Compiling Cilk++ Programs from the Command Line*** (see "Building, Running, and Debugging" Page 29) section described Cilk++ compile options. Options can also be set from within Visual Studio.

1. 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.
2. Expand the **Cilk++ Compiler** list in the **Configuration Properties** tree on the left.
3. Set the options as required.
4. 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.
5. To build the ***serialization*** (Page 155) of the Cilk++ program, which is especially useful for ***debugging Cilk++ programs*** (Page 37), expand the **General** list under **Cilk++ Compiler**. Then set **Compile as C++** to **yes**.

VISUAL STUDIO AND CILK++ PROGRAM EXECUTION

Just as when ***running Windows Cilk++ programs*** (Page 34) compiled from the command line, when running a Cilk++ program built with Visual Studio, the Cilk++ program will appear to consume all the resources of all the processors in the system, even when there is no parallel work to perform. This is not a problem, however, as described previously.

DEBUGGING CILK++ PROGRAMS

This section describes how to debug Cilk++ programs with the Visual Studio debugger or Linux's `gdb`. Before using these debuggers, however, several preliminary steps are helpful.

DEBUGGING INITIAL STEPS

If you are converting an existing C++ program, debug and test it as thoroughly as possible before converting to Cilk++. If you are writing a Cilk++ program from scratch, test the serialization. Serial bugs will impact both serial and parallel versions of the program, but the bugs are typically easier to detect and resolve in the serial version.

Next, find any race conditions. Some races may be obvious accesses to nonlocal variables; more subtle races may be discovered using the ***Cilkscreen Race Detector*** (Page 75). Resolve all races using one or more of the following techniques, as described in "***Data Race Correction Methods*** (Page 68)":

- Restructure the code
- Use a Cilk++ reducer
- Use a `cilk::mutex` lock, other lock, or atomic operation

DEBUGGING STRATEGIES

The principal difficulty in debugging a Cilk++ program is the multithreaded operation.

Windows Only: An additional problem with Windows debugging is that there is very limited direct support for debugging Cilk++ code. While you can set a breakpoint in a Cilk++ function or method, you cannot examine variables, and the stack may not display properly.

There are several strategies, however, that can mitigate these problems. These steps are required for **Windows** and recommended for **Linux**.

- *Use a serial build (the "serialization (Page 155)")*, which will compile code with the C++ compiler, creating a serialized C++ version of all Cilk++ keywords and calls. This will allow show all the state normally available in the debugger.
  - **Windows:** Command line builds can use the `/cilkp cpp` option, and Visual Studio users should refer to the "***Visual Studio Compiler Options*** (Page 36)" section to build a serialized Cilk++ program.

- **Linux:** See the end of the "***Building from the Linux Command Line*** (Page 29)" section for a description of the `-fcilk-stub` option.

- *Debug and Test the Serialization*. Run the program's serial version, which will have the same *serial semantics* as the original C++ program. Use any logging or other techniques to assure that the logic is correct. If the serial version does not run correctly, the parallelized version will not be correct either.
- *Limit use of Cilk++.* Cilk++ functions and methods can call out to C++ code. By limiting Cilk++ use to those functions and methods that need to use the Cilk++ keywords, you can continue to debug the C++ code normally. Note that the debugger may have trouble following the stack back through the Cilk++ code. The "***Mixing C++ and Cilk++*** (Page 115)" chapter describes the techniques for mixing languages.
- As with C++ programs, debugging Cilk++ programs is much easier if the program is not compiled with optimization. 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.

Windows Only: Stepping into a Cilk++ function will step through `__cilk_box()`. In a release build, these calls are inlined and optimized out by the compiler.

Chapter 6

CILK++ EXAMPLES

The Cilk++ distribution includes numerous examples to illustrate Cilk++ usage in a wide variety of situations. Each example is in its own directory in the `examples` directory, and each example directory includes the `.cilk` source code, Linux make file, and, for Windows, the Visual Studio 2005 solution and project files. Several directories contain a `ReadMe.txt` file with additional information about the example.

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

EXAMPLE DESCRIPTIONS

GENERAL EXAMPLES

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 85)" 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*** (see "Reducers" Page 85) and, like `matrix-transpose`, is a good platform to experiment with performance tuning.

wc-cilk

The `wc-cilk` ***example*** http://www.cilk.com/multicore-blog/bid/6899/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 with Cilk++ parallelism, 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 `cilk_for` grain size (see the ***Cilk++ Language Overview*** (Page 45) section), loop order, and more.

QUICKSORT EXAMPLES

The collection of quick sort examples illustrates basic Cilk++ techniques, including C++ conversion and dynamic link libraries. Other examples show a race condition, mutex usage, and a Cilk++ application with a Windows GUI.

qsort

The `qsort` example demonstrates how to speed up a ***Quicksort*** (http://en.wikipedia.org/wiki/Quicksort) algorithm with Cilk++. This is the same code as in the "***Getting Started*** (Page 19)" chapter.

qsort-cpp

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

qsort-race

The `qsort-race` example includes an intentional race condition. When run under the Cilkscreen Race Detector tool, 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 Specific: `qsort-dll` is a Visual Studio solution with three projects. It shows how to convert a Windows DLL (Dynamic Link Library) to Cilk++ (refer to *Converting Windows DLLs to Cilk++* (Page 121) 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 Specific: This example has a GUI interface built with MFC.

ASSESSING CILK++ EXAMPLE PERFORMANCE

Not only do these examples demonstrate Cilk++ conversion and programming techniques, but they also show the performance improvements that Cilk++ provides and the effect of changing the number of workers. For example, `qsort-cpp` is a C++ program, and `qsort` is the same program, converted to Cilk++, so you can directly compare their performance. Also, `qsort` allows command line specification of the worker count with the `-cilk_set_worker_count` option, so you can experiment with the number of worker threads and also test on systems with different core counts.

When running a Cilk++ program , it may appear that the program is consuming all the CPU resources. Despite appearances, this is not the case, as ***described earlier*** (see "Running Windows Cilk++ Programs" Page 34) for Windows.

CAUTION: In order to get repeatable performance comparisons, shut down any applications that are using CPU time, including background processes.

Chapter 7

CILK++ CONCEPTS

We introduce some key terminology here in order to clarify important concepts and to define the terms used to describe the Cilk++ language

There is a detailed Glossary at the end of this document.

The sections define and describe:

- Strands, spawning, and synchronization
- Program execution, parallelism, and dependency
- Scheduling, workers, and work stealing

STRANDS, SPAWNING, AND SYNCHRONIZATION

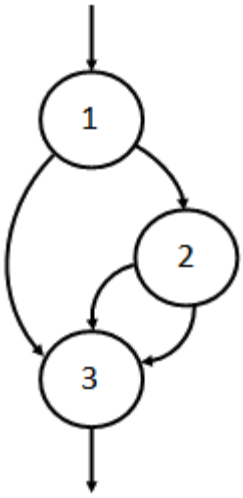
A ***strand*** (Page 155) is defined as a sequence of instructions without any parallel control structures. A typical serial program, therefore, consists of a single strand. When parallelism is introduced, multiple strands may execute in parallel. However, strands that *may* execute in parallel are not *required* to execute in parallel. The scheduler makes this decision dynamically in the Cilk++ runtime system, depending on several factors, including the ***worker*** (Page 156) count.

A strand ends and one or more new strands begin when a `cilk_spawn` (two new strands) or `cilk_for` (multiple new strands) statement is encountered. The strands end and join a new strand at a `cilk_sync` statement and when a `cilk_for` completes.

While strands are analogous to threads in some ways, strands describe the logical structure of a program, while Windows threads and Linux Pthreads are expensive, heavy-weight constructs that the operating system provides.

In the diagram below, strands are depicted as directed arcs, and two strands may execute in parallel. A strand depicted as an incoming arc at a node ends at that node. A strand depicted as an outgoing arc at a node starts at that node.

The incoming arc at node 1 is a strand. Node 1 then spawns a new strand, as implied by the two outgoing directed arcs. Another two strands are created at node 2. The sync at node 3, implied by the three incoming arcs, causes three strands to join. A single serial strand proceeds after node 3.



EXECUTION, PARALLELISM, AND DEPENDENCY

The *execution* of a Cilk++ program can be thought of as a directed acyclic graph (DAG), where each directed edge in the graph is a strand. The graph nodes (or "***knots***") are where the strand ends meet. If a knot has one incoming strand and two outgoing strands, it is a ***spawn knot***. If a node has multiple incoming strands and one outgoing strand, it is a ***sync knot***, and the incoming strands are synched with respect to each other at the sync knot. A Cilk++ program execution does not produce knots with both multiple incoming and multiple outgoing strands, nor does it produce a knot with a single incoming and single outgoing strand.

If there is a directed path from one strand **e1** to a second strand **e2**, the strands are ***in series***, and we say that **e1** ***precedes*** **e2** or **e2** ***depends on*** **e1**. In any scheduling of the Cilk++ program, **e1** must execute completely before **e2** can begin. If no path exists from **e1** to **e2** or vice versa, then there is no dependency between **e1** and **e2**, and the strands are ***in parallel***. Parallel strands may be scheduled to execute at the same time.

SCHEDULING, WORKERS, AND STEALING

Cilk++ uses a ***work stealing*** (Page 156) scheduler to distribute the strands that are ready to execute among the available cores. This scheduler is built upon operating-system ***thread*** (Page 155) software mechanisms. We call each operating-system thread participating in work stealing a ***worker*** (Page 156). By default, the Cilk++ runtime uses one worker thread per physical core. For example, on a system with two 4-core processors, the default for the Cilk++ runtime is to create 8 workers. Your application may choose to run with a different number of workers by making a Cilk library call or with a command line option, but the maximum speedup generally attainable is the smaller of the number of workers, the number of cores on the system, and the program's ***parallelism*** (Page 153).

Chapter 8

CILK++ LANGUAGE OVERVIEW

Cilk++ is a new language based on C++. This chapter describes the Cilk++ keywords, macros, and calling conventions and points out some small differences between the Windows and Linux implementations.

This chapter does not cover other Cilk++ system components, such as the ***Runtime System and Libraries*** (Page 61), the ***Cilkscreen Race Detector*** (Page 75), and ***reducers*** (Page 85).

CILK_SPAWN OVERVIEW

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 53)" section.

var is a variable with the type returned by *func*. It is known as the ***receiver*** (Page 154) 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 Race Detector*** (Page 75).

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 and Windows 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**: Won't compile; x is being constructed
- **Linux**: 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**: Won't compile; there is no receiver variable
- **Linux**: 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**: Won't compile — There is no receiver variable
- **Linux**: 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);
```

WINDOWS CILK_SPAWN RESTRICTIONS

This section is for ***Cilk++ for Windows*** programmers only

The Windows compiler currently requires a receiver for all non-void spawned functions, though this may change in the future.

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. Cilk Arts may lift this limitation 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.

EXCEPTION HANDLING

Cilk++ attempts to reproduce, as closely as possible, the semantics of C++ exception handling. This generally requires that Cilk++ reduce program 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.

Cilk++ for Windows: `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.

Cilk++ for Linux: `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. This behavior may change in a future Cilk++ release.

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

CILK_SYNC OVERVIEW

The `cilk_sync` statement means that the current function cannot run in parallel with its spawned children, as it is dependent on those 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 sync at the end of every function and every try block that contains a spawn. Cilk++ requires this feature 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 Cilk Arts blog, "***The Power of Well-Structured Parallelism (answering a FAQ about Cilk++)*** http://www.cilk.com/multicore-blog/bid/8092/The-Power-of-Well-Structured-Parallelism-answering-a-FAQ-about-Cilk", for more discussion of this point.

CILK_FOR OVERVIEW

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. A `cilk_for` loop is NOT simply a `for` loop with a `cilk_spawn` in each iteration; the latter would be inefficient for loops where the amount of work performed per iteration is small.

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 155)" 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 153), as that would cause a ***data race*** (Page 152). The ***Cilkscreen Race Detector*** (Page 75) will help detect these data races.

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 OPERATION

The iterations of a `cilk_for` loop may be executed in parallel; the control variable is not actually incremented repeatedly until the loop conditional expression tests `false`. Instead, the number of loop iterations is computed before the first iteration starts. To assist in this computation, Cilk++ requires that the control variable 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.

As a consequence of this behavior, `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`.

Similarly, Cilk++ cannot execute an infinite `cilk_for` loop such as:

```
cilk_for (unsigned int i = 0; i != 1; i += 0);
```

CILK_FOR LOOP LIMITATIONS

In order to parallelize a loop using the "divide-and-conquer" technique, the Cilk++ 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. Thus, 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.

CILK_FOR TYPE REQUIREMENTS

It is not necessary to read this section if there are no custom data types in the `cilk_for` statement. If there are any custom data types, you need to provide some methods to help the Cilk++ runtime compute the loop range size so that it can be divided.

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.
- Cilk++ 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 TUNING — GRAINSIZE

The `cilk_for` statement creates multiple spawn operations, with overhead associated with each spawn. In some situations, you can improve performance by reducing or increasing the number of spawns, and, consequently, reducing the overhead or increasing the program's parallelism. This is achieved by specifying the "grain size" using:

```
#pragma cilk_grainsize = expression
```

Before describing the pragma, here is an overview of `cilk_for` operation followed by some hints about selecting a good grain size:

- `cilk_for` divides the loop iterations into chunks to be executed serially. A chunk is a sequential collection of one or more loop iterations.
- There is an invisible `cilk_spawn` for each chunk.
- The maximum size of each chunk is called the "grain size", which specifies the maximum number of consecutive loop interations in the chunk.
- If the grain size is too small, then the `cilk_spawn` overhead will reduce performance.
- If the grain size is too large, then there will not be enough parallelism to keep the cores busy, and multicore performance will suffer.
- Cilk++ uses a default formula to calculate the grain size. The default works well under most circumstances.
- To override the default grain size, use the pragma.
- A pragma grain size of 0 results in the using the default grain size formula, and a negative value gives unpredictable results.

For example, if the grain size is k and you have the statement (assume for simplicity that n is a multiple of k):

```
cilk_for (int i = 0; i < n; ++i)
    MyFunc(i);
```

The statement executes as if the grain size were one and the loop were written as:

```
cilk_for (int j = 0; j < n/k; ++j)
    for (int i = 0; i < k; ++i)
        MyFunc(k*j + i);
```

As mentioned previously, the default grain size will usually work 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 the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) tool to estimate a program's ***work*** (Page 156), ***span*** (Page 155), and spawn overhead. This information can help determine the best granularity and whether it is appropriate to override the default grain size.

To specify the grain size, use the pragma:

```
#pragma cilk_grainsize = expr
```

where *expr* is any expression yielding a C++ integral type (including enums), such as `int` or `long`. The pragma should immediately precede the `cilk_for` statement to which it applies.

An example that attempts to balance these factors using the number of workers and the loop count is:

```
#pragma cilk_grainsize = n/(4*cilk::current_worker_count())
cilk_for (int i=0; i < n; ++i) MyFunc(i);
```

Several ***examples*** (see "Cilk++ Examples" Page 39) use the pragma. Specifically, see:

- `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 115)", shows how to call Cilk++ functions from C++ code. This is often useful in large applications.

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.

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:** This will produce a compiler or linker error.
  - **Linux:** If a program contains both `main()` and `cilk_main()`, Cilk++ for Linux will use `main()`.
- 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 115)" 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). Cilk++ for Linux will change the linkage of the derived class function to match the base class. The Windows compiler will report an error.
- **Windows:** 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 Cilk++ for Linux), all these Microsoft-specific calling conventions give a function C++ language linkage.

- **Windows:** 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:** 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 155).

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 118)" 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: 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 Cilk++ installer creates `.sys_include` files in known system include directories.

Linux: 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.

1. 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.
2. 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
3. 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.
4. 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.

PREDEFINED 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 10003, indicating compiler version 1.0.3.

`__cilkartsbuild`

Compilers provided by Cilk Arts define this macro to the unique internal build number. Programmers should not need to test this macro. The **Cilk++ for Linux** compiler will print the build number when run with the `-v` option.

`__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)
```

INTERACTIONS WITH OS FEATURES

Cilk++ applications must interact with the host operating system. This section describes interactions with OS threads. The first section deals with general threading issues, while the section discusses using MFC with **Cilk++ for Windows**.

OPERATING SYSTEM THREADS

Cilk strands are not operating-system threads. A Cilk strand will always be run by the same worker, and, therefore, the same OS thread for the strand's duration. 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, since the thread may change.
- Do not use operating system locks or mutexes across `cilk_spawn`, `cilk_sync`, or `cilk_for` statements, since only the locking thread can unlock the object. See the "***Holding a Lock Across a Strand Boundary*** (Page 71)" section.

MFC WITH CILK++ FOR WINDOWS

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

MFC (*Microsoft Foundation Classes* — used to create Windows user-interface components) depends upon thread local storage for mapping 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 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 a Cilk++ strand is not guaranteed to run on the UI thread, the first type of MFC application is the only kind that can use Cilk++. If the application is in the second form, convert it to the first form before attempting to convert the code to Cilk++. The steps are described next.

To convert to the first form:

1. To use Cilk++ in an MFC application, 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 child thread. The C++ program may already be in this form; if not, debug the C++ program to assure that the logic and thread management are correct.
2. Declare a `cilk::context` and call the context's `run()` function (***Cilk++ runtime functions*** (see "Runtime System and Libraries" Page 61)) from within this computation thread, creating the initial Cilk++ strand. The computation thread leaves the main (UI) thread available to run the message pump for processing windows messages and updating the UI.
3. Before terminating, the main (UI) thread should wait for the computation thread to complete, using `WaitForSingleObject()`.
4. Update the UI from within Cilk++ code by sending messages to the UI thread. Pass the handle (`HWND`) for the UI window into the computation strand (i.e., the strand, created by the computation thread, which is running Cilk++ code).
5. When the Cilk++ code needs to update the UI, it should send a message to the UI thread by calling `PostMessage`. `PostMessage` marshals and queues the message in the message queue associated with the thread that created the window handle. Do **NOT** use `SendMessage`, as `SendMessage` is run on the currently executing Cilk++ thread, which is not the correct (UI) thread.

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

Additional cautions:

- When the main UI thread creates the computation thread (assume this is done in a separate function), 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`, since the whole purpose is to avoid the MFC thread-local variables.

Chapter 9

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 Cilk++ 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 Cilk++ runtime uses separate stacks.

`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, Cilk++ will create a worker for every physical core on the system. ***Hyperthreaded processors*** (http://en.wikipedia.org/wiki/Hyper-threading) count 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 Cilk++ runtime, 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 explicit context and which takes an argument list.

In many cases, there is no need to be aware of the Cilk++ runtime, and there is no need to declare a context. `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 115).
- You must use C++ for the main program before executing Cilk++ code, as described in "***MFC with Cilk++ for Windows*** (Page 57)"

GETTING THE WORKER ID

The Cilk++ runtime 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 Cilk++ 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 alternate to `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()` usage, see `matrix-transpose` in the ***Cilk++ Examples*** (Page 39) collection. That example uses `cilk::run()` instead of `cilk_main()`, which the other examples use.

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*** ("data race" Page 152). The "***Locking and Nondeterminism Pitfalls*** (see "Locking Pitfalls" Page 69)" section describes potential problems with mutexes, such as ***deadlocks*** ("deadlock" Page 152) 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 the Cilkscreen Race Detector that it should consider some sequence of code protected by a "lock". The ***Race Condition*** (Page 154) 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 68)" section describes other locking mechanisms.

MISER MEMORY MANAGER

Some memory managers perform poorly when used by parallel applications such as Cilk++. Therefore, Cilk++ 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 two Cilk Arts blogs:

- "***Multicore Storage Allocation*** http://www.cilk.com/multicore-blog/bid/7904/Multicore-Storage-Allocation"
- "***Miser – A Dynamically Loadable Memory Allocator for Multithreaded Applications*** http://www.cilk.com/multicore-blog/bid/7812/Miser-A-Dynamically-Loadable-Memory-Allocator-for-Multi-Threaded-Applications". The blog contains 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 152)" 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

The following paragraphs are for ***Cilk++ 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.

This following paragraphs are for ***Cilk++ 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++.

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 Cilk++ and the default install location:

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

MISER LIMITATIONS

There are several general and Windows-specific Miser limitations.

- Once enabled, Miser cannot be disabled and will handle all C/C++/Cilk++ memory management requests 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.

This paragraphs are for ***Cilk++ for Windows*** programmers only.

- You cannot use the Windows `FreeLibrary()` function to free the `Miser.dll` module.
- The project must use the "Multithreaded DLL" "Runtime Library" "Code Generation" compiler option, `/MD` or `/MDd`. Miser cannot intercept calls to the static C/C++ runtime library.
- 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 10

RACE CONDITIONS

Race conditions (or simply "races") are a major cause of bugs in parallel programs. This chapter describes:

- Race condition theory and classification, including data races
- How to resolve data races
- Lock use and implementation
- Determinacy races that are not data races
- Pitfalls with locks and nondeterminism in general

The next two chapters describe the Cilkscreen Race Detector and reducers in much more detail.

DEFINITIONS AND RACE CONDITION TYPES

We'll define races in terms of the Cilk++ ***strand*** (Page 155). A ***race condition*** (Page 154) is a source of ***nondeterminism*** (Page 153) whereby the result of a concurrent computation depends on the timing or relative order of instruction execution in individual strands. Nondeterminism is the property of a program when it behaves differently from run to run when executed on exactly the same inputs.

The two principal race condition types that are of concern are:

1. ***Data Race:*** 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.
2. ***Determinacy Race:*** A race condition that occurs when two parallel strands access the same memory location and at least one strand performs a write.

A data race, then, is a special case of a determinacy race.

A determinacy race that is not a data race can occur even if parallel strands use locks properly when accessing a ***shared memory*** (Page 155) location; see the "***Determinacy Races That Are Not Data Races*** (Page 68)" section.

These definitions apply to shared memory, which is the memory model that Cilk++ assumes when dealing with multiple CPU systems. In general, the definitions could apply to shared files, databases, or other shared objects.

For a complete 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. The paper uses the term "general race" instead of "determinacy race".

DETERMINACY RACES THAT ARE NOT DATA RACES

There are determinacy races that are not data races. These races are not necessarily bugs; consider the application requirements to determine the appropriate action.

As a simple example, suppose that each `cilk_for` iteration creates an element and pushes it onto a list. This would be a data race on the list object.

If you were to add a `cilk::mutex` (see "cilk::mutex and Related Functions" Page 63) with a lock/unlock pair around the push operation, there would be no data race. However, there is now no assurance about the order of elements in the list. That is, there would be a determinacy race. The application's functional requirements would determine whether or not this is a bug.

Nonetheless, using a reducer would generally be a better solution, improving performance in this example, since the mutex would reduce parallelism. Furthermore, there would be no determinacy race.

DATA RACE CORRECTION METHODS

There are several ways to remove or fix a race condition, two of which use Cilk++ features.

- *Fix a program bug:* The `qsort-race` race is a bug in the program logic. The ***Cilkscreen Race Detector*** (Page 75) (RD) shows the program line numbers involved, so you can examine the data ranges used in the function call. The `qsort` function contains the correct code. Some race conditions are manifestations of underlying logic bugs.
- *Change the algorithm:* In some cases, a changed algorithm will eliminate a race without significantly affecting performance. As a simple illustration, consider the `cilk_for` example, which populates an array. Suppose you summed the values in order to compute the average. Summing inside the `cilk_for` loop would create a race condition, which is eliminated by summing the array elements at the loop end.
- *Use Cilk++ reducers (Page 85):* Several examples (`reducer`, `sum-cilk`, `hanoi`, etc.) illustrate reducer usage. Reducers maintain determinacy.
- *Use a cilk::mutex lock, other lock, or atomic operation:* See the next section for more information about lock usage. Even if locks are used properly to remove a data race, there may still be determinacy races, which RD will not detect. For this and other reasons described at the end of this chapter, lock usage is discouraged unless it is absolutely necessary; reducers are generally the preferred solution.

LOCKS AND THEIR IMPLEMENTATION

There are many synchronization mechanisms that may be implemented in the hardware or operating system. Additional hazards remain even after using a lock; refer to the ***Locking and Nondeterminism Pitfalls*** (see "Locking Pitfalls" Page 69) section. As mentioned previously, only use locks if they are absolutely necessary.

The Cilkscreen Race Detector (RD), described in detail in the next chapter, recognizes the following locking mechanisms; it does not recognize any others.

- Cilk++ provides the `cilk::mutex`, part of the Cilk++ ***Runtime System and Libraries*** (Page 61), to create critical code sections where it is safe to update and access shared memory or other shared resources safely. RD 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`.
- `cilk::fake_mutex` does not actually lock anything. Use this to suppress the RD race report if the reported race is benign.
- **Windows Specific:** ***Windows*** `CRITICAL_SECTION` http://msdn.microsoft.com/en-us/library/ms682530.aspx objects provide nearly the same functionality as `cilk::mutex` objects, and RD will not report races on accesses protected by the `EnterCriticalSection()`, `TryEnterCriticalSection()`, and `LeaveCriticalSection()` functions.
- **Linux Specific:** ***Posix Pthreads mutexes*** http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html (`pthead_mutex_t`) provide nearly the same functionality as `cilk::mutex` objects, and RD will not report races on accesses protected by the `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions.
- RD recognizes atomic hardware instructions such as those generated by compiler intrinsics by both the GCC and Microsoft compilers.

There are other operating system-specific mutexes, but these methods are nearly always slower than a `cilk::mutex`. Furthermore, RD 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 `CRITICAL_SECTION` (**Windows**) and `pthread_mutex_t` (**Linux**) objects.

LOCKING PITFALLS

There are numerous potential pitfalls when using locks and shared memory. This section briefly describes some common potential problems. These same problems can occur with any system that allows parallel operations, although the actual locking objects (to cite one example) would have different names and properties. The pitfalls are not unique to Cilk++.

As a general goal, avoid using locks whenever possible. If an alternate solution is applicable, such as using ***reducers*** (Page 85), use that alternate solution to improve performance and reliability.

LOCKS CAUSE NONDETERMINACY

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 nondeterminism, 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 nondeterminacy may not be a problem if the update is a ***commutative operation*** (Page 151), such as integer addition. However, many common operations, such as appending an element to the end of a list, are not commutative, and you cannot guarantee program execution results.

Floating point arithmetic is not commutative either because of rounding errors. "***Operations that MIGHT be Suitable for Reducers*** (Page 100)" describes a situation where are no data races, but there is a determinacy race in the sense that the results can vary from one run to the next.

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 REDUCE PARALLELISM

Parallel strands will not be able to run in parallel if they concurrently attempt to access a shared lock. Consider using a reducer if possible.

Nonetheless, if you must use locks, there 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 parallelizing. 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 11

CILKSCREEN RACE DETECTOR

The Cilkscreen Race Detector (RD) monitors Cilk++ program operation with test input. It identifies potential ***data race*** (Page 152)s encountered during execution with the test input, although it only detects data races in Cilk++ strands that are caused by Cilk++ constructs.

The RD helps to ensure reliability by alerting you to possible memory access conflicts within Cilk++ strands. Data races can be removed using one of the ***data race correction methods*** (Page 68) described previously.

To identify such races, invoke RD on a Cilk++ executable with carefully selected test input data cases, much as when running regression tests. You should run RD multiple times, once for each test case. Test cases should be selected so that you have complete code coverage and, ideally, complete path coverage. RD can only analyze code that is executed, and races may only occur under certain data and execution path combinations.

RD will run the binary executable program as a ***serial execution*** (Page 155) on a single worker, and it will monitor all memory reads and writes that are executed. When the program terminates, RD outputs information about races which are potential read/write and write/write conflicts. A race is reported if any possible scheduling of the executed reads and writes could produce results different from the serial program execution.

In addition to the race detector, Cilkscreen includes an additional feature, the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) (CPPA), discussed in a separate chapter.

RD detects data races, but it does not detect every ***determinacy race*** (Page 152); see the preceding "***Race Conditions*** (Page 67)" chapter.

CILKSCREEN RACE DETECTOR REPORTS AND USAGE PROCESS

The Cilkscreen Race Detector will report if there is any way to schedule a *lock-free* Cilk++ program with specified input data so that the results in memory would be different from the results of running the ***serialization*** (Page 155) of the program with the same input.

The lock-free condition is essential to the report accuracy as ***locks cause nondeterminacy*** (Page 70). There are some additional facts to bear in mind about the RD report:

- The RD implementation is based on a mathematically provably correct algorithm (for references, see "***Additional Resources and Information*** (Page 8)").
- Every reported data race should be investigated, although some reported data races may be benign. False positives will not be reported (but see the "***Cilkscreen Race Detector Limitations*** (Page 84)" section for an exception to this rule when using thread-local storage).

- If there are one or more data races, RD will identify and report at least one. It cannot identify all data races since the data race produces a nondeterministic result that could affect subsequent operation.
- As a consequence of the previous bullet, the correct process is to fix all reported data races and then run RD again, repeating until no races remain.
- RD does NOT report other determinacy races (see "***Determinacy Races That Are Not Data Races*** (Page 68)"); it only reports data races.

RD will ignore all threads except the one running the Cilk code. RD's analysis depends on two Cilk++ code features:

- Cilk++ has serial semantics. In other words, you'll get the same result if you execute the code in series or in parallel.
- The Cilk++ spawn/join model limits the scope of code running in parallel, and clearly delimits when strands join.

This implies an additional condition; RD only reports nondeterminacy caused by parallel Cilk++ strands. If, for example, you create a Windows or Linux native thread, and that thread modifies a global variable that is also modified in a Cilk++ strand, the data race will not be reported. Similarly, RD will not detect races between native threads and will not monitor the behavior of the Windows and Linux native thread management API.

Goal: The goal when writing a Cilk++ program is to create a deterministic, lock-free, program whenever possible.

CILKSCREEN RACE DETECTION AND TEST DATA

To reinforce the importance of the test data, remember that RD cannot identify every *potential* data race. RD detects only those races that can occur on the input provided. For example, given input **I** and a function **f**, if the program run on **I** never calls **f**, then **f** is not checked for races. There may be another input on which the program calls **f** and (potentially) manifests a race. Therefore, it is important to select test case input that provides good code coverage.

For example, consider this simple function which is spawned so that several strands execute the function in parallel.

```
static int g;
. . .
void spawned_function (int m)
{
  if (m < 10)
   g++;
}
```

The data race involving the global variable g will only be reported if you test the program with data that causes the function argument, m, to be less than 10. Otherwise, this function contains an undetected potential data race. Testing a program with RD is similar to any other program regression testing; the results can only be as good as your test data cases.

CILKSCREEN RACE DETECTOR OPERATION

RD monitors locks to ensure that races are not reported if memory accesses are protected by locks. Specifically, no race occurs if memory accesses that would cause a race occur while the accessing strands hold at least one lock in common.

The race detector works by monitoring all memory reads and writes during the program's binary execution. This strategy offers two key advantages:

- Race detection is performed on the application's production version, ensuring that there are no extra races reported or missed due to possible differences that would occur if a special build were required.
- RD will identify races that occur in any code called from a Cilk++ strand, including C or C++ code in your application, third-party libraries, and system runtime libraries.

Note: RD will only monitor for races within Cilk strands. Any races that occur in other threads, or before or after the call to `cilk::context::run` will be ignored. `cilk_main()` is run within a strand.

Consider the following code fragment, modified from the `reducers` example — see ***Example Descriptions*** (see "Cilk++ Examples" Page 39):

```
data_t arr[ARRAY_SIZE];
    ...
int accum = 0;

cilk_for (int i = 0; i < ARRAY_SIZE; ++i) {
    accum += do_work(arr[i]);
}
printf("accum = %d\n", accum);
```

As a C++ `for` loop, this is legitimate code. But as a `cilk_for` loop, there is a data race on the variable `accum`, because there are possible loop schedules for which the value of `accum` will be overwritten. RD will detect this race and print out the line number where the race occurs. The RD output is in the "***Cilkscreen Race Detector Output*** (Page 79)" section.

The race can be removed using a Cilk++ reducer as described previously. However, there may still be a race if a called function (such as `do_work()` in the code fragment above), is not thread safe. For instance, if `do_work()` modifies a global `static` variable, RD will report a race. The `reducer` example has a comment discussing this point.

CILKSCREEN DESIGN AND IMPLEMENTATION NOTES

- As mentioned previously, RD only analyzes Cilk++ code and functions called from Cilk++ code. Cilk++'s `cilk_spawn` and `cilk_sync` follow a strict fork/join paradigm which RD requires to keep its memory usage within acceptable bounds. Other threading technologies do not obey these constraints and cannot be analyzed by RD, which only detects races between Cilk++ strands.
- RD's dynamic monitoring causes the program to run slower. Tests suggest a slowdown of a factor of 15x to 30x normal execution time, although the slowdown can be considerably more or less. The slowdown is greatest when testing a debug build.
- RD testing is faster with input data that will cause the program under test to run faster. For example, you might test a sort routine with a small data set. However, be careful not to reduce the code coverage.
- RD requires several times the memory footprint of the original application. Cilk Arts is working on some strategies to mitigate the memory overhead scope. If you encounter problems due to memory overhead, please contact **support[at]cilk[dot]com** for assistance and recommendations.
- RD recognizes memory management operations (`malloc()`, `free()`, `calloc()`, `new`, etc.). Therefore, if Strand 1 allocates, reads, writes to, and frees a memory unit, and a later allocation in Strand 2 (which runs in parallel with Strand 1) uses some of the same memory addresses, RD will (correctly) not report this as a race.
- RD recognizes lock constructs as listed in the "***Locks and Their Implementation*** (Page 68)" section.

CILKSCREEN RACE DETECTOR EXECUTION

RD runs on binaries built in any mode, but it only reports file and line numbers for binaries for which debug information is available. If debug information is not available, RD reports the addresses and files of the instructions that caused the race condition.

At least one of the reported addresses must involve a write. The addresses may be the same if a line in a spawned function performs a read-modify-write sequence, such as `i++`.

It is best to use a debug version for ease in locating races that are found. Also, set compiler options to turn off function inlining (see "***Cilkscreen Race Detector Limitations*** (Page 84)").

Once you have removed all data races, be aware that different compiler optimizations will change code paths and may introduce or hide races. Therefore, be sure to perform a final RD test with your production code to ensure that there are no additional races in the production, as opposed to debug, code.

To use RD from the command line:

1. Build the application with the necessary options to produce debug information. For best results, build without code optimization.

2. Go to the directory containing the executable and invoke:
   `cilkscreen [`*cilkscreen options*`] `*your_program*` [`*program options*`]`
3. The "***Cilkscreen Race Detector Command Line Options*** (Page 81)" section shows how to specify RD options.
4. RD runs the program with the arguments supplied and writes information about any races it detects either to `stderr` or to a specified file. If the executable was built with debug information, then the output includes file and line information about races it detects. Use the line number information to find the races in the code.

Linux Specific: RD auto-detects whether the analyzed program is a 32-bit or 64-bit binary. Therefore, you can run RD on either a 32 or 64-bit system, regardless of the analyzed program's nature.

Windows Specific: From Visual Studio:

1. Open the Cilk++ project in Visual Studio and build it.
2. Select **Tools -> Run Cilkscreen Race Detector** to run the program under RD. RD runs the program with the command arguments in the project debugging properties.
3. RD's results appear in a Visual Studio Cilkscreen window after the program exits and reports detected data races, along with line numbers. If the application was built with debugging information, double-clicking on a line of output opens the source file in Visual Studio and positions on the source line.
4. RD creates a `.csl` file logging any detected races. Select **Tools -> Open Race Detector Log** to view this or any other `.csl` log file.

Resolving Races: Refer to the "***Data Race Correction Methods*** (Page 68)" section.

CILKSCREEN RACE DETECTOR OUTPUT

RD reports source code line numbers that are involved in races; that is, the lines containing the conflicting memory accesses. There are two line numbers, sometimes the same, for each race, representing the conflicting accesses. At least one access must be a write.

The sample output here is from the `qsort-mutex` example, where all lines related to locking and unlocking are commented out. This leaves a single line, Line 60:

```
partition_count++;
```

Line 60 involves both a read and a write access to `partition_count` without a lock to protect the access. RD reports line 60 twice, along with the call sequence. RD also reports the numerical `partition_count` memory location (`0043F5CC`) but cannot report the variable name.

The RD command, using just four data points, is:

```
cilkscreen qsort-mutex 4
```

The **Windows** output is shown below. The **Linux** output is slightly different, but the line numbers provide the essential information.

```
Sorting 4 integers

Race on location 0043F5CC between
>qsort-mutex.cilk:60: qsort-mutex.exe!sample_qsort<int *>+0xc4
(eip=0041D19C)
and
>qsort-mutex.cilk:60: qsort-mutex.exe!sample_qsort<int *>+0xbc
(eip=0041D194)
>qsort-mutex.cilk:66: qsort-mutex.exe!sample_qsort<int
*>+0x19f (eip=0041D277)
  called from here
>qsort-mutex.cilk:84: qsort-mutex.exe!qmain+0x1e9
(eip=0040335D)
  called from here
>qsort-mutex.cilk:110: qsort-mutex.exe!cilk_main+0xa5
(eip=004087CD)
  called from here
cilk10.dll!__cilkrts_ltq_overflow+0x137 (eip=100081C7)
  called from here
Guilty Spawn:
>qsort-mutex.cilk:65: qsort-mutex.exe!sample_qsort<int
*>+0x12f (eip=0041D207)

Race on location 0043F5CC between
>qsort-mutex.cilk:60: qsort-mutex.exe!sample_qsort<int *>+0xc4
(eip=0041D19C)
and
>qsort-mutex.cilk:60: qsort-mutex.exe!sample_qsort<int *>+0xc4
(eip=0041D19C)
>qsort-mutex.cilk:66: qsort-mutex.exe!sample_qsort<int
*>+0x19f (eip=0041D277)  called from here
>qsort-mutex.cilk:84: qsort-mutex.exe!qmain+0x1e9
(eip=0040335D)
  called from here
>qsort-mutex.cilk:110: qsort-mutex.exe!cilk_main+0xa5
(eip=004087CD)
  called from here
cilk10.dll!__cilkrts_ltq_overflow+0x137 (eip=100081C7)
  called from here
Guilty Spawn:
>qsort-mutex.cilk:65: qsort-mutex.exe!sample_qsort<int
*>+0x12f (eip=0041D207)
0.093 seconds
```

```
Sort succeeded.
2 errors found by Cilkscreen
Cilkscreen suppressed 1 duplicate error messages
```

Other interesting RD output information includes:

- The number of duplicate error messages. With this example, the number increases with the number of data points. For example, 8 data points yields 6 duplicates.
- The number of errors (2), which is not a function of data size in this example.
- The two reported races involve the same line numbers but differ in terms of the instruction locations, which are highlighted.
- The call trace through lines 60, 66, 84, 110, and 65.

Cilk++ for Windows Only: CRD race information will also list the name of the variable involved in the race if: 1) You built the program with debug information, and 2) the code with the race is in a C++ function (that is, a function with C++ linkage) which is called from Cilk++ code.

CILKSCREEN RACE DETECTOR COMMAND LINE OPTIONS

The command line format is:

```
cilkscreen [cilkscreen-args] [--] command [command options]
```

1. The optional `args` are listed next. These options apply to the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) tool (CPPA) as well as to the RD.
2. `command` represents the Cilk++ program, with options, that Cilkscreen will analyze. "`--`" can be used to separate the command from the Clkscreen options, but it's not necessary.

"`cilkscreen-args`" is zero or more of the following options to control the output content, format, and location. Unless specified otherwise, output goes to the `stderr` (the console by default).

`-r logfile`

The ASCII output showing detected races is sent to the named file.

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

`-x logfile.xml`

The output showing detected races is in XML format in the named file. 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`

Vebose debugging output, showing detailed trace information such as DLL loading. Unless the `-l` option is specified, the debug output will be written to `stderr`.

`-l tracefile`

The ASCII trace information is sent to the named file. 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.

`-h`, `-?`, and no args or command

Display Cilkscreen usage.

`-v`

Display version information and exit.

`-w`

Run the ***Cilkscreen Parallel Performance Analyzer*** (Page 103).

For example, get verbose output of `qsort-race` sorting 100 values as follows:

```
cilkscreen -d -- qsort-race 100
```

To get the race information in a file, run:

```
cilkscreen -r logfile.txt qsort-race 100
```

CILKSCREEN RACE DETECTOR: EXAMPLES WITH DATA RACES

Two examples with data races are:

1. `qsort-race`, one of the Quicksort Examples. The race exists because two spawned recursive function calls use overlapping data ranges.
2. ***Reducer Usage — a Simple Example*** (Page 85), 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.

RD detects both these data races. It is easy to create data races (also detectable with RD) in some other examples. For instance, the `reducer` example would have a race if you changed the declaration inside the `dowork()` function to "`static volatile int j = 0;`". The `sum-cilk` example would have a data race if you replaced the `sum_max` reducer with a shared variable.

CILKSCREEN RACE DETECTOR ADVANCED FEATURES

Disable/Enable Instrumentation: RD causes the program it is testing to run much slower than it otherwise would because it monitors all memory reads and writes. Program segments that are C++ may not need to be checked for the races introduced by Cilk++ parallelization. The following functions are provided to disable and re-enable RD:

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

Disable/Enable Checking: You can turn off race checking in a more limited sense within Cilk++ code:

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

When checking is disabled, RD does not record any memory reads or writes until checking is enabled again.

Note that these calls to RD do not cause any performance penalty when the application is not run under the RD because they are used to place markers in the executable code. There is no actual call at runtime.

Caution: `__cilkscreen_disable_checking()` decrements a counter and `__cilkscreen_enable_checking()` increments the same counter; the counter is initialized to 0 and should never become positive. The calls should be balanced, analogously to parentheses. Consequently:

- A call to
  `__cilkscreen_enable_checking()`
  prior to any call to
  `__cilkscreen_disable_checking()`
  will cause a fatal runtime error since the counter would become positive.
- If there are two calls to
  `__cilkscreen_disable_checking()`
  followed by a single call to:
  `__cilkscreen_enable_checking()`
  there will be no checking in the subsequent code. Another
  `__cilkscreen_enable_checking()`
  call would be required to enable checking.

Clean Memory: Some applications use techniques such as "look-aside lists" or "suballocators" to reduce memory management overhead. RD may report races on memory allocated with these techniques.

Inform RD about program-specific memory managers by using `__cilkscreen_clean()` to declare that a memory block is "clean" — not currently owned by any thread. Specify the memory range beginning and end before writing to "newly allocated" memory or passing it to other parts of the program.

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

As with other RD calls, `__cilkscreen_clean()` is implemented in a way that incurs no runtime cost when the application is not run under RD.

A potential concern is memory you allocate and free; for example suppose you call `malloc()` to allocate some memory, use that memory, free it and then call `malloc()` again. Would there be a problem if `malloc()` returned the same memory address? RD recognizes calls to Miser and system memory management functions such as `malloc()`, so it is not necessary to call `__cilkscreen_clean()` before freeing memory.

CILKSCREEN RACE DETECTOR LIMITATIONS

The current RD version has two known limitations and shortcomings.

- *Inlined functions.* Inlined functions can cause a confusing call stack. The entire body of an inlined function is reported as the line at which it is inlined.
- *Thread-local storage*. RD does not understand about thread-local storage and will report parallel use of thread-local storage as a race, even though this may not be a problem. In general, using thread-local storage in a Cilk++ program requires a solid understanding of how Cilk++ works. If you are certain the TLS usage is safe, use fake locks to suppress the RD race report.
- RD does not recognize any lock objects other than those listed in "***Locks and Their Implementation*** (Page 68)". Therefore, RD might report a false positive if you use some other lock implementation.

Chapter 12

REDUCERS

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

Cilk Arts provides ***reducers*** to address the problem of accessing nonlocal variables in parallel code. See "Reducers: Introduction to Avoiding Races" for a simple example, and see two Cilk Arts papers, ***"Are Determinacy-Race Bugs Lurking in YOUR Multicore Application?"*** (http://www.cilk.com/multicore-blog/bid/5254/Are-Determinacy-Race-Bugs-Lurking-in-YOUR-Multicore-Application) and ***"Global Variable Reconsidered"*** (http://www.cilk.com/multicore-blog/bid/5672/Global-Variable-Reconsidered) provide additional examples.

Conceptually, a reducer is a variable that can be safely used by multiple strands running in parallel. Cilk++ 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. Cilk++ 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.

Cilk Arts provides a reducer library, and you can write a custom reducer based on the examples in this chapter.

REDUCER USAGE — A SIMPLE EXAMPLE

A simple, and common, reducer application is to accumulate a sum in parallel. Consider the following serial program that calls a `MyFunc()` function for each loop index value and accumulates the answers into the `result` variable:

```
int MyFunc(int i);

int main()
{
    // ...
```

```
    int result = 0;
    for (std::size_t i = 0; i < N; ++i)
    {
        result += MyFunc(i);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```

Changing the `for` to a `cilk_for` causes the loop to run in parallel, but doing so creates a data race on the `result` variable. Change `result` to a `reducer_opadd` hyperobject to resolve the race.

```
#include <reducer_opadd.h>
int MyFunc(int i);

int cilk_main()
{
    // ...

    cilk::hyperobject<cilk::reducer_opadd<int> > result;
    cilk_for (std::size_t i = 0; i < N; ++i)
    {
        result() += MyFunc(i);
    }
    std::cout << "The result is: " << result().get_value()
              << std::endl;
    return 0;
}
```

The changes in the serial code show the steps required to use any reducer:

1. Include the reducer's header file.
2. Declare the result variable as a `reducer_opadd<int>` rather than an `int`.
3. Permit the loop iterations to operate in parallel by changing the `for` loop to a `cilk_for` loop.
4. Because the reducer is an object, change `result` to `result()` to get and update a dynamic view of the reducer.
5. Retrieve the reducer's terminal value with the `get_value()` method after the `cilk_for` loop is complete.

Two ***examples*** (see "Cilk++ Examples" Page 39), `reducer` and `sum-cilk` use `reducer_opadd` in the same manner as in this code fragment.

Reducer Caution: 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.

CILK ARTS REDUCER LIBRARY

The Cilk++ reducer library contains the following reducers, including `reducer_opadd`, which was used in the preceding example.

For additional usage examples, see the comments in the header files (in the installation's `include` directory).

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

REDUCER/HEADER	ID; UPDATE	DESCRIPTION
`reducer_list_append` `<reducer_list.h>`	empty list; `push_back()`	Creates a list using an append operation. Note that 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.
`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_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.

REDUCER/HEADER	ID; UPDATE	DESCRIPTION
`reducer_opadd` `<reducer_opadd.h>`	`0;` `+=, =, -=, ++, --`	Performs a sum.
`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; `+=`	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; `+=`	Provides a shorthand for a `basic_string` of type `char`.
`reducer_wstring` `<reducer_string.h>`	Empty string, or arguments to constructor; `+=`	Provides a shorthand for a `basic_string` of type `wchar`.

REDUCER USAGE — ADDITIONAL EXAMPLES

Examples here show how to use several Cilk++ reducers, including those in the Cilk++ reducer library. The examples help illustrate reducer features.

Also remember that the header files for the reducers (such as `reducer_opadd.h`) contain usage examples in the comments. The header files are in the installation `include` directory.

REDUCERS — A STRING EXAMPLE

`reducer_string` builds 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 Cilk++ runtime 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 assure serial semantics, the serial semantics are only assured at the end of the calculation, such as at the end of a `cilk_for` loop, after Cilk++ 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.

This rule about never calling `get_value()` within a loop will be intentionally violated in a later example (***Writing Reducers - A "Holder" Example*** (see "Writing Reducers — A "Holder" Example" Page 93)). In that example, the reduce operation does nothing, and `get_value()` returns a valid local view used only within the loop 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::hyperobject<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
```

REDUCERS — A LIST EXAMPLE

`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::hyperobject<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;
}
```

REDUCERS — A TREE TRAVERSAL EXAMPLE

The previous reducer examples all performed the update operations within a `cilk_for` loop, but reducers are just as powerful in other contexts, such as recursively traversing a data structure (tree, list, etc.). During the traversal, collected value information is added to a list. There is a similar example in the article: ***Global Variable Reconsidered*** (http://www.cilk.com/multicore-blog/bid/5672/Global-Variable-Reconsidered).

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

cilk::hyperobject<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);
}
```

REDUCERS — A TEMPLATE CLASS EXAMPLE

If you define a template class and use that class as the type when declaring a reducer, it is necessary to assure that the template class has ***C++ linkage*** (see "Cilk++ and C++ Language Linkage" Page 53). Therefore, use `extern "C++"` with the class definition.

Here is the skeleton of the class definition for "`Addable`" (a user defined class):

```
extern "C++"
{
    template <class T>
    class Addable {
    public:
        Addable& operator+=(const T& v) {
            d_value += v;
            return *this;
        }
        ... Other addition methods, constructor(s),
            destructor(s)
    private:
```

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

`Addable` can now be used with `reducer_opadd` or other appropriate reducers:

```
cilk::hyperobject<cilk::reducer_opadd<Addable> > result;
```

Without the `extern "C++"`, this declaration would not compile.

REDUCER DEVELOPMENT

You can develop a custom reducer if the Cilk Arts Reducer Library does not satisfy your requirements.

Reducers must implement the following two methods:

- A *default constructor* – This will be called to create a new reducer instance the first time the reducer is referenced in a strand. The default constructor should initialize the reducer to the *identity value* for the operation the reducer implements and also initialize any other state variables. The default constructor signature for a reducer named `reducer_name` is:

  ```
  reducer_name()
  ```

  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. Identity value examples include:

  - `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 ""
    ```

- A *reduce method* that merges this reducer instance with another instance. The operation the `reduce` method implements must be an ***associative operation*** (see "About the Glossary" Page 151), but does not need to be a ***commutative operation*** (Page 151). The application program never calls the `reduce` method directly.

  The `reduce()` signature requires a single parameter, which is a pointer to the specific reducer:

  ```
  void reduce(reducer_name * right)
  ```

If implemented correctly, the reduce methods will retain the reducer's serial semantics. That is, the results of running the application serially, or using a single worker, is the same as running the application with multiple workers. The runtime, together with the `reduce` method's associativity, assures that the results will be the same, regardless of strand execution and completion order.

All other methods implemented by a reducer provide the API that updates the reducer's stored value or get its value.

REDUCE/UPDATE DISTINCTION

The reduce and update operations are closely related, but there are important distinctions.

"Update" is the API your reducer supplies to perform whatever operations are appropriate, such as appending an element to a list. A reducer can have multiple update operations, such as ++, +=, and `-=` in a summing reducer. In some cases, such as a ***holder reducer*** (see "Writing Reducers — A "Holder" Example" Page 93), there will not be any update operations.

Cilk++ programs use the update API.

The need for distinguishing the update API from the `reduce` method can be seen by considering a `cilk_for` loop where each loop iteration appends (an update operation) an element to a list. Cilk++ will generally perform multiple consecutive loop iterations in a single strand, with each strand resulting in lists built with update operations. As the strands complete, Cilk++ transparently uses `reduce` to combine the individual lists into larger lists, in the correct serial order.

The operations can be much more complex, or simpler, than arithmetic sums or lists.

REDUCER DEVELOPMENT EXAMPLES

Cilk Arts provides the ***Cilk Arts Reducer Library*** (Page 87), but Cilk++ programmers can build custom reducers suited to the needs of their data structures and applications.

The patterns illustrated in the Cilk Arts reducer header files (`reducer_list.h`, etc.) in the `include` directory of the Cilk++ installation are possible models, although those examples are complex and may not be useful; try the simpler examples here. Two ***Cilk++ examples*** (Page 39) also contain 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.

Note that reducer design is currently in flux and may change, perhaps significantly, in the future.

WRITING REDUCERS — A "HOLDER" EXAMPLE

This example shows how to write a reducer that is nearly as simple as possible, since it does not have any update methods, and the `reduce` method does nothing.

Such a reducer has a practical use; suppose there is a global temporary buffer used by each `for` loop iteration. This is safe in a serial program but not in a parallel program. The following Cilk++ example is even simpler as it consists of a single variable value that acts as a "holder" for the values used within strands.

```
static int temp;
    . . .
void MySwap (int *a, int *b)
{
    temp = *a;
    *a = *b;
    *b = temp;
    return;
}

void reverse (int * A, int n)
{
    cilk_for (int i = 0; i < n/2; ++i)
    MySwap (&A[i], &A[n-i-1]);
    return;
}
```

There is a race on the global variable, `temp`, but the serial program would work properly. In such a simple example, it would be easier to declare `temp` inside `MySwap()`. However, many existing C++ programs have such constructs with larger global data structures or buffers, and it would not be easy to restructure the program.

The solution is in two parts: first, the reducer, called `temp_buff`, and, second, the modified `MySwap()` function. There is no need to change the `reverse()` function.

The default constructor and reduce methods are called by the Cilk++ runtime, so the reducers must have C++ linkage, not ***Cilk++ linkage*** (see "Cilk++ and C++ Language Linkage" Page 53). Therefore, enclose the class definition in an `extern "C++"` construct.

```
// A Holder Reducer, holding a single integer.
// Reducers are always C++
extern "C++" {
class temp_buff
{
public:
        // REQUIRED default constructor, initializes to 0.
    temp_buff() : t_value(0) { }
        // REQUIRED reduce method.
        // Merges left (this) and right instances
    void reduce(temp_buff* right) {
        // Does nothing.
    }
        // API for this reducer
        // get_value - returns the current value
    int get_value() const { return t_value; }
        // assignment operator - sets the value
    int& operator= (const int &v) {
```

```
        t_value = v;
        return t_value;
    }
private:
    int t_value;
};
}  // extern "C++"
```

The modified `MySwap()` function is:

```
cilk::hyperobject<temp_buff> temp;
    . . .
void MySwap (int *a, int *b)
{
    temp() = *a;
    *a = *b;
    *b = temp().get_value();
    return;
}
```

This `temp_buff` reducer provides a strand-local instance of type `int` (it would be easy to generalize this to use a template):

- The default constructor will create the instance with the first reducer reference.
- The reduce method does nothing. Since we only use `t_value` for temporary storage, there is no need to combine the two instances.
- The default destructor will invoke the destructor for `t_value`, freeing its memory.
- The only API provided is `get_value()`, which returns the data the reducer holds, and assignment.
- Since the local view value produced in a loop iteration is not combined with values from other iterations, it is valid to call `get_value()` to get the local view within a `cilk_for` loop using `MySwap()`.
- The identity value, 0, is actually arbitrary in this example; any integer value could be used. There is no reduce operation, so there is no identity.

WRITING REDUCERS — A SUM EXAMPLE

This example shows a more complex, custom, reducer with 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 ***Addable template class*** ("Reducers — A Template Class Example" Page 91) from a previous example. 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 includes multiple update operations.

```
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 required `reduce()` method combines the results of the left (`this`) subsequence with the right (`other`) subsequence.
- The identity value is provided by the default constructor.
- The *update* operations are provided by the += and ++ operators.
- We could have added update variations such as `-=` and `--`, provided that 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 be useful in any way.
- Only the default constructor and `reduce` methods have names and argument lists that are prescribed by the `cilk_reducer` template.

REDUCERS — WHEN ARE THEY NEEDED AND POSSIBLE?

This section describes a straight-forward "thought experiment" to determine if a reducer is appropriate and feasible in a given situation. Some of this material repeats earlier points in order to combine, reinforce, and generalize the considerations required for designing and writing a reducer.

Look for a pattern in which multiple values (of some type, `T`, for instance), are computed in multiple individual steps, and the values are combined, perhaps into a container, using an update operator, `op`. `id` is the "identity" value for type `T` and operation `op`.

Pattern 1: Loops

First, consider the ***previous example*** (see "Reducer Usage — A Simple Example" Page 85) which sums integers produced by a function. `+=` is the update operation, and `0` is the identity element.

```
int MyFunc(int i);
int main()
{
    // ...

    int result = 0;
    for (int i = 0; i < N; ++i)
    {
        result += MyFunc(i);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```

We've already shown how to use `reducer_opadd` in this example, but suppose that the application uses a different update operation, `op`, on integers (the example could easily be generalized to data type `T`) and that `id` is the identity element for `op` (0 for addition and 1 for multiplication are examples). That is, `a op id ==` a for any integer value, `a`. The code comparable to the arithmetic sum example is:

```
int *MyFunc(int i, ...);
// . . .

int L = id;
for (int i=0; i<N; ++i) {
    L = L op *MyFunc(i, ...);
}
```

As a thought experiment, ask if this computation could be arbitrarily separated into distinct loops and the results combined with an associative reduce operator, `rop`, as follows:

```
int *MyFunc(int i, ...);
// . . .

int L1 = id;
for (int i=0; i<k; ++i) {
    L1 = L1 op *MyFunc(i, ...);
}
int L2 = id;
for (int i=k; i<N; ++i) {
    L2 = L2 op *MyFunc(i, ...);
}
int L = L1 rop L2;
```

If you could organize the computation in this way so that the results are correct for any data values and any value of `k`, `0 <= k < N`, *and* `rop` is associative, then you could parallelize the computations using `cilk_for`. A reducer would be needed to avoid a race. There may be a useful reducer in the Cilk++ library; alternatively, write a custom reducer.

Pattern 2: Recursive Divide-and-Conquer

Also consider whether it is possible to replace the two loops with recursive function calls which return their results in function arguments rather than function values. This is the approach used in the `hanoi` example program. The original serial program might have the following form, where `MyRecFunc` (assume all arguments are passed by reference) updates variable `L` of type `T`:

```
T L;             // Global variable

MyRecFunc (x, y) {
    if (base_case(x, y)) { L = L op value (x, y);}
    else {
        MyRecFunc (x1, y1);
        MyRecFunc (x2, y2);
    }
}
  . . .
L = init_value;
MyRecFunc(a, b);
```

The question to ask, then, is if there is a reducer operation, `rop`, so that we can add a function parameter and write the code as follows:

```
T L;             // Global variable

MyRecFunc (x, y, T LP) {
    if (base_case(x, y)) { LP = value (x, y);}
    else {
        T L1 = id, L2 = id;
        MyRecFunc (x1, y1, L1);
        MyRecFunc (x2, y2, L2);
```

```
            LP = L1 rop L2;
        }
    }
       . . .
    MyRecFunc(a, b, L);
```

Again, refer to the `hanoi` example program to see how to implement this technique.

REDUCER OPERATION REQUIREMENTS

Combining the previous information, reduce operation requirements are as follows:

- The `reduce` operation must have an identity and must be associative, meaning that `((a rop b) rop c)` gives exactly the same value as `(a rop (b rop c))`. The reason is that the `cilk_for` execution will cause individual values to be grouped differently, and the loop will be executed in multiple separate loops (not necessarily just two).
- The `reduce` operation does not need to be commutative since reducers assure that the individual values are combined in exactly the same order as in the serial version. This is essential for operations such as string building and appending to the end of a list (where list order is important).

Mathematically, the `reduce` operation, identity value, and set of operands form a ***monoid*** http://en.wikipedia.org/wiki/Monoid.

For example, suppose a `cilk_for` loop has 12 iterations, producing values `L0, ..., L11`. The final value, `L`, could be computed in many different sequences, due to the nondeterminism of `cilk_for` execution, showing why associativity is required but `rop` does not need to be commutative. Three of many possible computation sequences are:

```
L = (L0 rop L1 rop L2) rop (L3 rop L4 rop L5) rop
    (L6 rop L7 rop L8) rop (L9 rop L10 rop L11)
L = (L0 rop L1) rop (L2 rop L3 rop L4 rop L5) rop
    (L6 rop L7 rop L8) rop (L9 rop L10 rop L11)
L = (L0 rop L1 rop L2 rop L3 rop L4) rop
    (L5 rop L6 rop L7 rop L8 rop L9 rop L10 rop L11)
```

Summary: If the loop can be executed in multiple steps, such as loops or recursive function calls, to produce a sequence of values, and if the sequence can be combined in order with an associative operation, then the computations can be run in parallel and use a reducer.

OPERATIONS SUITABLE FOR REDUCERS

Numerous operations, useful in a wide variety of applications, are associative and have an identity element. Cilk++ library reducers support several basic operations.

- **Arithmetic addition and multiplication**, including matrix addition and multiplication. This statement applies to most arithmetic data types other than floating point, which is not strictly associative due to rounding errors
- **Maximum and minimum** values
- **Appending to the end** (or beginning) of a list, deque, or string
- **Boolean operations** such as `and`, `or`, and `xor`. This applies to operations on Boolean vectors and matrices as well
- Many operations used to define an abstract mathematical object such as a **group, ring, or monoid**

OPERATIONS THAT MIGHT BE SUITABLE FOR REDUCERS

Other operations *might* be suitable; evaluate the requirements and expectations.

An important example is ***floating point addition***, which is not associative, and results can vary widely depending on the order in which large and small numbers are combined. The results with the same input data could vary from one execution to the next. The problem is aggravated when adding or multiplying floating point arrays. A very simple example is to test if a single precision sum is positive. The first combination is positive, whereas the second is zero.

```
(-1.0 + 1.0) + 0.00000001
-1.0 + (1.0 + 0.00000001)
```

In many practical situations, the serial execution has no specific meaning relative to any other, and these variations are acceptable and expected. A reducer would be appropriate. Furthermore, you may be able to determine bounds on the variation due to the data's dynamic range or the use of double precision arithmetic.

The `sum-cilk` example sums floating point (single precision) numbers, and there are small, but detectable differences from one run to the next when using large values for the matrix size (about 10,000) and different grain sizes.

OPERATIONS NOT SUITABLE FOR REDUCERS

There are some operations that are not associative and therefore would not be suitable for a reducer. However, in some cases, a reducer could be used if it's possible to relax the requirements for "equality" when determining if the operation is associative.

- **Objects with Time Stamps**. Suppose the operation appends objects to a list, and each object has a "creation" time stamp. Furthermore, the timestamps are required to be monotonically increasing with the order in the list. Parallelizing the object creation will not work in this situation, even though the objects will occur in the correct list order. The timestamp order cannot be assured. This situation is due to parallelization, not the reducer or non-associativity.
- **Subtraction** is not associative. That is, `a - (b - c) != (a - b) - c`. However, you could work around this problem by summing negative values.
- **Exponentiation** is also not associative. That is, `a ^ (b ^ c) != (a ^ b) ^ c`.

- **Conditional Addition**, where the arithmetic is not continuous. For example, consider computing a bank balance by adding deposits (positive numbers) and withdrawals (negative numbers), and the bank allows negative balances but charges $10 if the balance is negative. This `rop` operation is not associative. For instance,

```
(100 rop 50) rop -60 != 100 rop (50 rop -60)
```

REDUCERS WITH COMPLEX STATE

The examples and the Cilk++ library reducers have been simple in the sense that they compute just a single object value, such as a sum or a list. `reducer_max_index` and `reducer_min_index` are exceptions since they compute both an index and a max (or min) value. However, it would be sufficient to compute just the index and then find the value from the array.

In other situations, you can use separate reducer objects since the computed state values can be separated. For example, when summing vectors, can just have a separate `reducer_opadd` object for each dimension. Or, when computing the mean and variance of a set of numbers, one `reducer_opadd` object could be used to sum the values and another to sum the squares of the values.

However, if you are forming the product of a set of complex numbers or a set of matrices, then the entire set of values representing the product complex number (two values) or matrix would need to be retained in a single reducer.

Chapter 13

CILKSCREEN PARALLEL PERFORMANCE ANALYZER

The Cilkscreen Parallel Performance Analyzer (CPPA) is a simple program parallelism profiler. By profiling a Cilk++ program during its serial execution, CPPA summarizes program parallelism and provides speedup estimation. This tool is designed to help to answer an important question that can significantly affect the parallel program performance – "Does my program have sufficient parallelism?".

CPPA computes program ***parallelism*** (Page 153) as the ratio of the ***work*** (Page 156) to the ***span*** (Page 155) when the program is run with a specific input. Note that for some programs the parallelism measure can vary widely, depending on the input.

The following sections will first illustrate the information provided by the CPPA profiler and then use examples to describe how to interpret the information to identify performance problems.

USING THE CILKSCREEN PARALLEL PERFORMANCE ANALYZER

Run CPPA to analyze a program by using the `cilkscreen -w` option, followed by the program name and options. For instance, analyze the qsort example program with 100000 data points with the command:

```
cilkscreen -w qsort 100000
```

Use the same ***Cilkscreen Command Line Options*** ("Cilkscreen Race Detector Command Line Options" Page 81) as are available with the Cilkscreen Race Detector. Thus, to send the report to a file, use:

```
cilkscreen -r cppa_report.txt -w qsort 100000
```

The report, in file `cppa_report.txt`, using Cilk++ for Linux, is:

```
Cilkscreen Parallel Performance Analyzer V1.0.3, Build 6960
1) Parallelism Profile
   Work :                                  53924334 instructions
   Span :                                  16592885 instructions
   Burdened span :                         16751417 instructions
   Parallelism :                           3.25
   Burdened parallelism :                  3.22
   Number of spawns/syncs:                 100000
   Average instructions / strand :         179
   Strands along span :                    83
   Average instructions/strand on span :   199914
   Total number of atomic instructions :   18
```

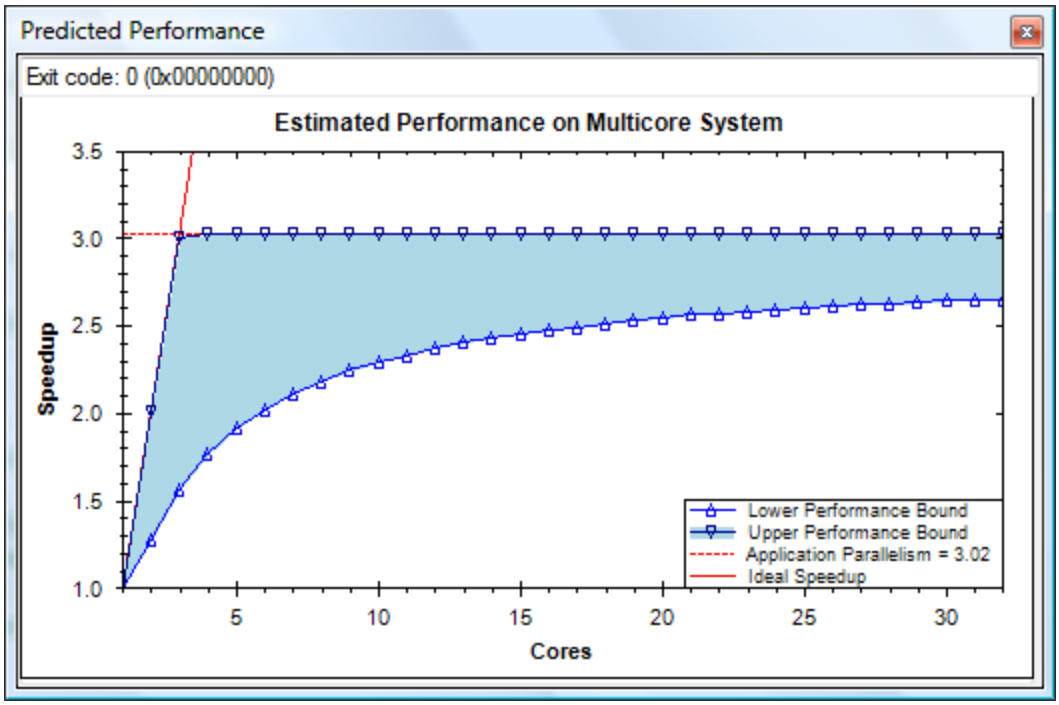
```
2) Speedup Estimate
   2 processors:          1.31 - 2.00
   4 processors:          1.55 - 3.25
   8 processors:          1.70 - 3.25
   16 processors:         1.79 - 3.25
   32 processors:         1.84 - 3.25
```

The next section interprets this output.

Windows Specific: You can execute CPPA from Visual Studio (not available with Visual Studio 2008 Express):

1. Open the Cilk++ project in Visual Studio and build it.
2. Select **Tools -> Run Cilkscreen Parallel Performance Analyzer** to run the program under the CPPA. CPPA runs the program with the command arguments specified in the project debugging properties.

CPPA results appear in a Visual Studio window after the program exits and displays a graph of estimated speedup ("Lower Performance Bound") as a function of core count. The parallelism (or "Upper Performance Bound") is the horizontal top line in the graph. The results are slightly different from the Cilk++ for Linux results.



Get a copy of the detailed report (similar to the command line version) by right clicking on the graph and selecting "Show Run Details" from the context menu.

CPPA creates a `.dsw` file logging any detected races. Select **Tools -> Open Parallel Performance Analyzer Log** to view this or any other `.dsw` log file; you will see a display such as the example above.

The actual numbers will be different using Cilk++ for Linux and Cilk++ for Windows 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.

INTERPRETING THE PROFILING REPORT

The CPPA Cilk++ Program Profiling Report has two sections showing the statistics collected while executing the program with the specified input:

- The **Parallelism Profile** output includes two important pieces of program information: ***parallelism*** (Page 153) and **burdened parallelism**, along with counts of instructions, strands, spawns, syncs, and atomic operations.
- The **Speedup Estimation** section provides an estimated program speedup range on 2, 4, 8, 16 and 32 processor cores. The speedup range shows the lower and upper bounds. The upper bound is determined by the minimum of the program parallelism and the number of processors used to run the program. The lower bound accounts for the runtime and scheduling overhead as a combined effect of the program structure and number of processors. A speedup lower bound less than 1 indicates that the program may slowdown instead of speedup when run on more than one processor.

MORE ABOUT THE PARALLELISM PROFILE

Program parallelism is the program's intrinsic parallelism measured by the program's work divided by its ***span*** (Page 155). Since the span is the theoretically fastest execution time for a parallel program when run on an infinite number of processors, parallelism is the theoretical upper bound of a program's speedup on an infinite number of processors. Therefore, we should not expect a program to have speedup greater than its parallelism.

This parallelism measurement does not consider any possible runtime and scheduling overhead from spawn, sync, scheduling, and so on that may affect the speedup. We introduce another metric called "Burdened Parallelism", which accounts for the estimated overhead. The burdened parallelism is computed as the ratio of work and burdened span, where the burdened span is the program's span plus the estimated runtime and scheduling overhead.

The burdened span is an approximate upper bound on the program execution time on an infinite number of processors. The burdened parallelism is an approximate lower bound on program speedup using an infinite number of processors.

The CPPA also collects and reports some other statistics, including the number of spawns and syncs in the program execution.

CPPA PROGRAM SEGMENT REPORTS

Running CPPA under the Cilkscreen tool provides information about the complete program. This information may include significant code that will not be parallelized, such as initialization and output. Frequently, you will want to analyze individual program segments, such as loops and functions, that can be parallelized and where you want to focus attention.

The Cilk++ `workspan` object allows you to generate CPPA statistics for code segments rather than a complete program. There are three `workspan` functions:

- `start()` - Resets all counters in the object and enables counting (instructions, spawns, etc.).
- `stop()` - Disables counting. Counters are not reset.
- `report(`*stream*`)` - Write the report using data collected between the preceding `start()` and `stop()` calls. *stream* is a C++ output stream, such as `std::cout`.

Here is an example code segment surrounded by `workspan` function calls, followed by the output.

```
workspan ws;
cilk::hyperobject<cilk::reducer_opadd<float> > sum_max;
 . . .
ws.start();
#pragma cilk_grainsize = expression;
cilk_for(int j = 0; j < nn; ++j)
    sum_max() += column_max(A, nn, j);
ws.stop();
std::cout << std::endl << "CPPA Interesting segment." <<
std::endl;
ws.report(std::cout);
 . . .
```

Here is representative output that `report(`*stream*`)` generates, which shows information from the Parallelism Profile:

```
CPPA Program Segment Report:
  Work = 12508663
  Span = 14266
  Burdened span = 73944
  Parallelism = 876.816
  Burdened parallelism = 169.164
  #Spawns = 999
  #Atomic instructions = 1001
```

You will get this output only if you run the program under the CPPA, using the command line:

```
cilkscreen -w
```

If you are not running CPPA, then the `workspan` `start()`, `stop()`, and `report()` functions are disabled, and there is no significant performance impact.

A CPPA `workspan` object can analyze multiple segments since each `start()` call resets the counters. Alternatively, you can use several `workspan` objects.

PERFORMANCE IMPROVEMENT WITH CPPA

When a Cilk++ program's speedup is less than expected, the CPPA provides information to help understand the possible performance problems. The CPPA diagnosis requires three steps, explained in the following sections.

1. Check the program's parallelism
2. Check the program's burdened parallelism and the speedup estimation
3. Check other common performance limitation causes

The following sections and the "***Performance Issues in Cilk++ Programs*** (Page 111)" chapter cover some common causes of performance problems in parallel applications. Specifically, these sections show how to exploit the CPPA to help improve Cilk++ program performance.

CHECK THE PROGRAM'S PARALLELISM

Parallelism is an upper bound of the program's speedup.

For example, if a program's parallelism is less than 2, the program should not be expected to run more than 2 times faster than a serial execution no matter how many processors are used.

There could be many factors that cause low program performance, such as:

- Poor parallel algorithms or small granularity that leads to high overhead. Use the CPPA information to help identify the cause of limited speedup.
- A small data sample; run with representative data. For instance, the example `matrix_multiply_dc_notemp` run with an 8x8 matrix has parallelism of approximately 1 because the program executes serial loops on such a small matrix. Larger matrices, such as 64x64, have increased parallelism. Likewise, a `cilk_for` loop with a limited range may not have significant parallelism.

If a program does not achieve speedup close to its parallelism when using multiple workers and cores, then proceed to check the program's burdened parallelism.

CHECK THE PROGRAM'S BURDENED 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.

A program with large parallelism but small burdened parallelism may indicate small strand granularity. The average number of instructions per strand in the parallel profile report could be a good indicator. If a program has less than a few hundred instructions per strand, the scheduling overhead may become a significant portion of total execution time.

The following code shows an example program that repeatedly operates on an array. Since the work can be executed in parallel for different array elements, it uses `cilk_for` to parallelize the work over different array elements.

```
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 negative speedup. That is, running this program on 4 processors takes about 3 times longer than serial execution.

A segment of CPPA output for this code is listed below. 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 granularity is the problem.

```
. . . . . .
1) Parallelism Profile
   . . . . . .
   Span:                          2281596648 instructions
   Burdened span:                 32281638648 instructions
   Parallelism:                   3.06
   Burdened parallelism:          0.22
   . . . . . .
   Average instructions / strand:          698
```

```
   Strands along span:                      5000006
   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
```

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 (that is, increase the recursion threshold that is used in the `matrix` and other examples).
- Replace spawns of small tasks with serial calls.
- Designate small "leaf" functions as C++ rather than Cilk++ functions, reducing calling overhead.
- Use the `cilk_for` ***grainsize*** ("cilk_for Tuning — Grainsize" Page 52) to control granularity. CPPA uses the default grainsize unless there is a pragma to override the default.

For this particular example, we can revise the `repeat_work()` function to combine the repeated small work on each element into a larger task and parallelize execution of the larger task. Make this change by moving the `cilk_for` from the inner loop to the outer loop.

The revised program, shown below, achieves almost perfect linear speedup on 4 processors. If we look at the CPPA results with this modification, the parallelism and burdened parallelism have improved to 4.00.

```
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 CPPA report for the revised code.

```
1) Parallelism Profile
   . . . . . .
   Span:                            1359597788 instructions
   Burdened span:                   1359669788 instructions
   Parallelism:                     4.00
   Burdened parallelism:            4.00
   . . . . . .
```

```
Average instructions / strand:          258885669
Strands along span:                     11
Average instructions / strand on span:  123599798
```

CHECK OTHER COMMON PERFORMANCE PROBLEMS

If a program has sufficient parallelism and burdened parallelism but still doesn't achieve good speedup, the performance could be affected by other factors. See the "***Common Performance Pitfalls*** (Page 112)" section in the "***Performance Issues in Cilk++ Programs*** (Page 111)" chapter.

Chapter 14

PERFORMANCE ISSUES IN CILK++ PROGRAMS

The previous chapter showed how to use the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) to increase and exploit program parallelism. However, parallel programs have numerous additional performance barriers, as well as performance improvement opportunities.

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, including Cilk++ applications.

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 Cilk Arts blog, "***Making Your Cache Go Further in These Troubled Times*** http://www.cilk.com/multicore-blog/bid/6934/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 `example_get_time()` (defined in the `examples/include` directory) 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** ("cilk_for Tuning — Grainsize" Page 52). 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. Cilk++ uses 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** ("Locks Reduce Parallelism" Page 71). Locks generally reduce program parallelism and therefore affect performance. The current CPPA tool does not account for lock contention when calculating parallelism. Lock usage can be analyzed using other performance and profiling tools such as ***Intel VTune*** http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.
- **Cache efficiency and memory bandwidth** (Page 113). See the next section.
- **False sharing** (Page 113). See the section later in this chapter.
- **Atomic (Page 151) operations**. Atomic operations, provided by OS 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. The CPPA 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 Cilk Arts blog "***Making Your Cache Go Further in These Troubled Times*** http://www.cilk.com/multicore-blog/?Tag=cache" or the `matrix` and `matrix_multiply` examples cited in the "***Optimize the Serial Program First*** (Page 111)" section.

A simple way to identify bandwidth problems is to run multiple copies of the serial program simultaneously. 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 all other tested 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++.

FALSE SHARING

False sharing (Page 152) 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. The CPPA 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.

Chapter 15

MIXING C++ AND CILK++

This chapter provides extended documentation of C++ and Cilk++ application code integration, which is particularly important in large programs.

A common problem is to add parallelism to a large C++ program without converting the entire program to Cilk++. A first approach might be to "Cilkify" (that is, convert to Cilk++) entire classes. 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.

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 Cilk runtime each time through. The `qsort-dll` example (**Windows Only**), uses this approach.

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

In many cases, this is not necessary, and there is a fourth approach where Cilk++ calls C++. Just cilkify the key loop or recursive function. 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++: FOUR APPROACHES

There are four approaches to cilkifying a project:

1. Cilkify the entire project
2. Cilkify only the call tree leaves where the Cilk keywords are used, requiring that C++ functions call Cilk++ functions
3. Some combination of these two approaches
4. 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 Cilkifying a coherent module with a small number of public entry points.

For **Windows** examples using the second method (cilkify the call tree leaves), see "***MFC with Cilk++ for Windows*** (Page 57)" (also, the `QuickDemo` example) and "***Converting Windows DLLs to Cilk++*** (Page 121)" (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` ("cilk::context" Page 61) (or `cilk::run` (Page 62)) 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 Cilk++ and C++. Use the `__cilkplusplus` pre-defined 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 118)" 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 55)" 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 cilkified and others not. There are two ways to handle this:

1. Create a separate `.cilk` file for the Cilk++ code.
2. Convert the entire `.cpp` to `.cilk`, then wrap the non-Cilk++ 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 Cilk++ features, 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 C++ or Cilk++ file that is part of a serialization of a Cilk++ program is being compiled. 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 the Cilk library API. 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**) and `cilk++` (**Linux**) 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 16

CONVERTING WINDOWS DLLS TO CILK++

This chapter is for ***Cilk++ for Windows*** programmers only.

The ***Getting Started*** (Page 19) 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 uses Cilk++.

This section uses an example based on the previous `qsort` example to illustrate the conversion steps. The 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 Cilk++ library.
- `qsort-client` does not use `cilk_main()`, and, therefore, you cannot use the –`cilk_set_worker_count` option.

DLL CONVERSION INITIAL STEPS

Using the `qsort` conversion process used in ***Getting Started*** (Page 19) as a guide, the initial `qsort-cpp-dll` DLL conversion steps are:

1. Open the `qsort-cpp-dll` solution.
2. Select and expand the `qsort-cpp-dll` project within the solution.
3. Right click on the `qsort-cpp-dll.cpp` source file and convert it to Cilk++.

DLL SOURCE CODE CONVERSION

The next steps are required to modify `qsort-dll-cpp.cpp` to create a Cilk++ context and to 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, and the different steps are explained after the code.

```
19  /*
20   * An DLL quicksort implementation in Cilk++.
21   */
    . . .
29  #include <windows.h>
30
31  #include <cilk.h>
32  #include "qsort.h"
33
34  static cilk::context ctx;
35
36  BOOL WINAPI DllMain( HMODULE hModule,
37                       DWORD  ul_reason_for_call,
38                       LPVOID lpReserved)
39  {
40      switch (ul_reason_for_call) {
41          case DLL_PROCESS_ATTACH:
42              ctx.set_worker_count(0);  // Default count
43              break;
44
45          case DLL_PROCESS_DETACH:
46              break;
47
48          case DLL_THREAD_ATTACH:
49              break;
50
51          case DLL_THREAD_DETACH:
52              break;
53
54          default:
55              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:

1. Include the Cilk++ Header File: See Line 31.
2. 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.
3. 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.
4. Rename the `qsort` function: See Line 81. This function has Cilk++ linkage, and it should have a different name form the `sample_qsort` function called by the client application (see Line 83).
5. 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::cotext::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).

6. Write `cilk_sort_wrapper`; see Lines 68-81. Convert the parameters to the form that `cilk_qsort` requires.

The global Cilk++ context variable, `ctx` (Line 34) is constructed statically when `qsort-client` starts. Consequently, ***Cilkscreen*** (see "Cilkscreen Race Detector" Page 75) 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.

WINDOWS DLL HEADER FILE

The last step requires a change the header file, `qsort.h`, which will be used by both the client and the DLL projects. `sample_qsort` requires `extern "C++"`. This has no impact on the client project but is required in the DLL project. Here is the complete listing.

```
 1  #ifndef _QSORT_H
 2  #define _QSORT_H
 3  #ifdef _WINDLL
 4  #define QSORT_DLL __declspec(dllexport)
 5  #else
 6  #define QSORT_DLL __declspec(dllimport)
 7  #endif
 8
 9  extern "C++"
11      void QSORT_DLL sample_qsort(int * begin, int * end);
10  #endif
```

Chapter 17

LIMITATIONS AND RESTRICTIONS

This version of Cilk++ has several limitations, restrictions and known bugs, which may be removed in future releases. In most cases, limitations are not listed here if they were specified previously.

LANGUAGE ISSUES

General and OS-specific language issues are:

- Cilk++ does not support speculative parallelism. There is no mechanism to abort parallel computation.
- The `cilk_for` construct supports limited forms of loops, as described in the ***cilk_for overview*** `(Page 49)` section.

Linux Specific Issue:

- `cilk_for` may not be used in a constructor or destructor. It may be used in a function called from a constructor or destructor.

Windows Specific Issues:

- Pointers to members of classes with virtual base classes (data or functions) cannot be used as Cilk++ function arguments.
- Methods returning methods returning pointers to data members will not work properly.
- Secure SCL is explicitly disabled by the `cilkpp` program. Secure SCL adds additional code for debugging STL iterators.
- The Microsoft VC++ compiler will accept some constructs that are not part of the C++ standard. In some rare cases, the Cilk++ compiler will not accept the nonstandard construct.

PRODUCT ISSUES

- Performance optimization is not complete.
- Reducer syntax may change in a future release; the changes will be designed to simplify reducers and hyperobjects.

CILK++ AND THE C++ BOOST LIBRARIES

Using the ***Boost C++ Libraries*** http://www.boost.org/ in a Cilk++ program requires care. Cilk++ 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, install Boost outside of `/usr/include` and other standard include directories which are assumed to contain C and C++ code.
- On Windows, do not create a `.sys_include` file in the Boost include directory.

Cilk Arts is working on better support for Boost and similar styles of template programming.

OPERATING SYSTEM INTEGRATION

- The Cilk++ runtime may schedule a code sequence on a different thread after a `cilk_spawn`, `cilk_sync` or `cilk_for` statement. This has a number of implications for your code:
  - When using Windows thread-local storage (TLS) or Linux Pthreads thread-specific data in Cilk++ code, you cannot depend on being on the same thread after crossing a strand boundary.
  - C/C++ code that is called from Cilk++ code cannot depend on receiving multiple calls on the same thread.
  - Cilk++ code should not hold a lock when it executes a `cilk_spawn`, `cilk_sync`, or `cilk_for`.
- Cilk++ does not yet support functions with aligned frames and try-blocks. A frame may become aligned because it uses certain SSE instructions that address 16-byte chunks of memory that must be aligned on 16-byte boundaries. Cilk++ does support these aligned frames, and it does support try-blocks, but it does not yet support both in the same function.

Windows Specific: MFC uses Thread Local Storage for internal state. MFC should not be called directly from Cilk++ code. Instead, MFC applications should create a thread to run Cilk++ code and use `PostMessage` to pass information to MFC UI threads. See the section "***MFC with Cilk++ for Windows*** (Page 57)".

WINDOWS VISUAL STUDIO INTEGRATION

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

- 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 37) for details and workarounds.
- Cilk++ 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 the Cilkscreen Race Detector and Cilkscreen Parallel Performance Analyzer 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. The results will depend on the application being run. Some applications may terminate due to unrecognized arguements. Note that this is a bug in the integration of the Cilkscreen Race Detector and Cilkscreen Parallel Performance Analyzer with Visual Studio. The command line version of these tools handles redirection correctly.

Chapter 18

BUILDING CILK++ FOR LINUX

You can download the Cilk++ for Linux source code. Although we have combined the components into a single tar file for convenience, different components are released under different license terms. Be sure to understand the license applicable to each component before using or redistributing it. Detailed license information is provided in the "***License*** (Page 137)" chapter.

The source distribution file will have a name in the form:

```
cilk_src_6514.tar.gz
```

"6514" is the build number in this example. Place this file in a convenient location, go to the directory where the source is stored and enter the command:

```
tar -xvf path/cilk_src_6514.tar.gz
```

The source will be in the new subdirectory, `cilk_src`. Change the working directory to `cilk_src`.

The `gcc` subdirectory, exclusive of `gcc/cilk`, includes the Cilk++ compiler. The Cilk++ compiler is a version of GCC 4.2.4 which we modified in 2008 to support the Cilk++ language. Changes to `gcc` to support Cilk are generally marked as such — preceded by `#if TARGET_SUPPORTS_CILK` or with a comment or variable name indicating that the code is for Cilk. If you have received a binary copy of the product from Cilk Arts, the contents of the `gcc` directory satisfy the obligation under the GNU General Public License (GPL) to provide the corresponding source code. This source code is subject to the GPL or the GNU Lesser Public License (LGPL). These licenses are in `gcc/COPYING` and `gcc/COPYING.LIB` as well as in the "License" chapter.

Cilk currently requires a 32 or 64-bit x86 processor with SSE2 instructions. Configuring `gcc` for any other target will not enable Cilk.

`gcc/gdb` contains a tar file with the modified `gdb` source code. We made two changes to `gdb 6.8`.

- We disabled a check in `gdb/frame.c` that caused `gdb` to be confused by Cilk frames.
- We recognized the mangled form of the type of a pointer to a Cilk function.

The `gcc/cilk` subdirectory is not part of `gcc` and is not subject to the GPL. `gcc/cilk/cilk-mode.el` provides an experimental Emacs mode for editing Cilk++ code. `gcc/cilk/build` is the script we use to build the compiler.

The runtime directory includes the Cilk++ runtime. This is distributed under the terms of the Cilk Arts Public License, a copy of which is available in the LICENSE file in the top level directory of the Cilk source distribution and is in the "License" chapter. Note that the CAPL is different from the GPL. An explanation of the CAPL may be found on the ***Cilk Arts web site*** http://www.cilk.com.

The include directory contains header files defining the Cilk++ API, utility classes that may be useful with Cilk++, and "stub" files that may be included in Cilk++ code to allow compilation with an ordinary C++ compiler.

The top level of this source distribution also contains a build script. It will invoke "`build`" or "`make`" in subdirectories as appropriate. Arguments to the script are the source, build, and install directories. All three directories must be different (`gcc` cannot be built in the source directory). The source directory must be an absolute pathname, not "`.`" For example,

```
% mkdir /var/tmp/cilk-build
% ./build `pwd` /var/tmp/cilk-build /usr/local/cilk
```

The contents of the build directory (`/var/tmp/cilk-build` in the example above) are not needed after the build is complete. The compiler, runtime, and other files will be installed into subdirectories under the installation directory. For example, if the install directory argument is `/usr/local/cilk`, the compiler will be installed as `/usr/local/cilk/bin/cilk++` and the 64-bit runtime as

```
/usr/local/cilk/lib64/libcilkrts.so.1
```

Chapter 19

HOW CAN I ... ? COMMON QUESTIONS ABOUT USING CILK++

This chapter gives brief answers to common questions about using Cilk++ features. Many answers give step by step instructions with links to Programmer's Guide chapters and sections as well as external references.

Install Cilk++

1. Verify that you have satisfied the hardware and software requirements for either Cilk++ for Linux ("***Linux Installation*** (Page 11)") or Cilk++ for Windows ("***Windows Installation*** (Page 15)").
2. Install Cilk++ for your system using the instructions in those chapters.
3. Verify that Cilk++ is properly installed by building and running an example, as described next.

Run a Cilk++ Example

1. Read the "***Getting Started: Running Cilk++ Examples*** (Page 19)" section.
2. Select an example program, such as `qsort`.
3. Build and run the program as described. If you need more information, see the "***Building, Running, and Debugging*** (Page 29)" chapter.
4. Experiment with different input values and with different worker counts to see the effect on program performance.
5. Try other examples, such as `matrix_multiply` or `sum-cilk`, or any other example you choose.

Learn Cilk++ Language Basic Features

1. Read the "***Getting Started*** (Page 19)" chapter which introduces the three Cilk++ keywords and reducers.
2. Convert the `qsort-cpp` example to Cilk++, as described in "Getting Started".
3. View the code in some other examples and examine how Cilk++ keywords are used.

Learn Cilk++ Language Advanced Features

1. Read the "***Cilk++ Concepts*** (Page 43)" chapter.
2. Read the "***Cilk++ Language Overview*** (Page 45)" chapter.
3. Work with additional examples.

4. Modify the examples; for example, change `cilk_for` loops to try different data types for the control variable and different allowable methods to increment or decrement the control variable.

Convert an Existing C++ Application to Cilk++

1. Read the "***Getting Started*** (Page 19)" chapter, which shows the steps to convert an existing program to Cilk++.
2. Determine where to specify parallelism with `cilk_spawn` and `cilk_for` statements.
3. Test the Cilk++ program and measure its performance on multicore systems with different worker counts.
4. Additional steps, described later, include using the Cilkscreen Race Detector and the Cilkscreen Parallel Performance Analyzer tools.

Learn More about the Examples and See Additional Examples

1. The example programs are described briefly in the "***Cilk++ Examples*** (Page 39)" chapter.
2. There are `ReadMe.txt` files included with some of the examples; each file describes the example and its implementation. In some cases, there are suggestions for potential program improvements.
3. The Cilk Arts web site "***For Developers Only*** http://www.cilk.com/resources-for-multicoders/for-developers-only/" page has links to additional examples and more information about the "***Cilk++ Examples*** (Page 39)", such as performance graphs.

Determine Where to Use Cilk++ Keywords to Parallelize a Program

1. Determine the "hot spots" and time-consuming parts of your program. There are many useful tools to help in this task; the ***Intel VTune Performance Analyzer*** http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm is one such tool.
2. Be certain that your program uses good algorithms and that it is well-optimized for single core operation.
3. Determine if the program hot spots have code that can be parallelized.
4. Consider converting `for` loops to `cilk_for` loops. If you have nested loops, it is almost always best to convert the outer loop.
5. Recursive function calls that can run in parallel with the caller can be invoked with `cilk_spawn`; `qsort` is a well-known example.

Learn to Use the Cilkscreen Race Detector

1. Read the "***Cilkscreen Race Detector*** (Page 75)" chapter.
2. Run the race detector on the `qsort-race` example as described in the "***Cilkscreen Race Detector Usage*** ("Cilkscreen Race Detector Execution" Page 78)" chapter. Then run the `qsort-mutex` example, modified to contain a race, and comment out the lock calls, as described in the same chapter.
3. Cilk++ for Windows users can run the race detector both from the command line and from Visual Studio.

4. Examine and interpret the output for both examples as described in the "***Cilkscreen Race Detector Output*** (Page 79)" section.

Remove Races From a Cilk++ Program

1. Read the "***Race Conditions*** (Page 67)" chapter to learn more about races and their causes.
2. Use the Cilkscreen Race Detector tool to be certain that you have identified all data races and their root causes.
3. Determine which of the "***Data Race Correction Methods*** (Page 68)" is best for your situation. It may be necessary to learn more about reducers or locks (there's more on this later).
4. Examine the code carefully for any remaining potential determinacy races and correct them. Also, be aware of the potential "Locking and Nondeterminism Pitfalls".
5. Test the corrected program.

Use the Cilk++ Runtime System and Libraries

1. Read the "***Runtime System and Libraries*** (Page 61)" chapter to learn about the runtime features.
2. Determine what runtime features would be useful in your application. The chapter discusses where you would typically use the features.
3. As appropriate, look at the examples mentioned in the chapter to see how the runtime is used.

Use a Reducer

1. Review the introductory "Reducers: Introduction to Avoiding Races" section.
2. Read the "***Reducers*** (Page 85)" chapter up to, but not including, the "***Reducer Writing Examples*** ("Reducer Development Examples" Page 93)" section.
3. Look at the usage examples and compare them to the example programs. Look carefully at the reducer declaration, update, and `get_value()` operations. Be sure to understand the underlying associative operation and identity value.
4. Determine if you can solve your problem using a reducer in the "***Cilk Arts Reducer Library*** (Page 87)".
5. Otherwise, write your own reducer (more later).

Write a Reducer

1. First, understand how to use a reducer.
2. Read the "***Reducers*** (Page 85)" chapter starting at the "***Reducer Writing Examples*** ("Reducer Development Examples" Page 93)" section.
3. Determine the "reduce" and "update" operations for your reducer.
4. Also, determine the identity element and the underlying "monoid" (example, the set of integers with addition or the set of strings with concatenation).

5. Reread "***When is a Reducer Needed and Possible*** ("Reducers — When are they Needed and Possible?" Page 97)" to confirm that a reducer is appropriate in this situation.
6. Write the reducer, referring to the examples as needed. Additional, but more complex, examples are the reducer library implementations (see the "***Cilk Arts Reducer Library*** (Page 87)" section).

Learn How to Use the Cilkscreen Parallel Performance Analyzer

1. Read the "***Cilkscreen Parallel Performance Analyzer*** (Page 103)" (CPPA) chapter.
2. Run the CPPA on the `qsort` example as described in the chapter and interpret the results. Try this with several different inputs.
3. Run several other examples. `matrix_multiply`, `matrix_multiply_dc_notemp`, and `sum-cilk` demonstrate a variety of techniques. Run these, and other, examples with different input.
4. Cilk++ for Windows users can run the CPPA both from the command line and from Visual Studio.
5. Run the CPPA on your own Cilk++ programs; this may help you to improve performance.

Tune and Improve Cilk++ Application Performance

1. Read the "***Application Performance Improvement Techniques*** ("Performance Improvement with CPPA" Page 107)" section.
2. Read Cilk Arts blogs such as "***Finding Performance Bottlenecks & Data Races*** http://www.cilk.com/multicore-blog/bid/7454/Finding-Performance-Bottlenecks-Data-Races" and "***Making Your Cache go Further in These Troubled Times*** http://www.cilk.com/multicore-blog/bid/6934/Making-Your-Cache-Go-Further-in-These-Troubled-Times". You will find valuable tips and information.
3. Run the ***Cilkscreen Parallel Performance Analyzer*** (Page 103) to determine the parallelism in your program with typical input data.
4. Read the "`cilk_for Tuning`***: Grainsize*** ("cilk_for Tuning — Grainsize" Page 52)" section and consider setting the grain size rather than using the default grain size.
5. If you use a recursive divide-and-conquer algorithm, consider using a "recursion threshold" as in `matrix_multiply`, `matrix_multiply_dc_notemp`, and other examples. Use CPPA to analyze with different values and also measure performance.

Convert Large Applications to Cilk++

1. Determine if it is feasible or appropriate to convert all C++ code to Cilk++. Code that cannot run in parallel does not necessarily need conversion.
2. Read the "***Mixing C++ and Cilk++*** (Page 115)" chapter.
3. Determine if any of the "***Mixing C++ and Cilk++: Four Approaches*** (Page 115)" is appropriate for your application.
4. If you need to call Cilk++ code from C++ code, use the techniques shown in the chapter and demonstrated in examples cited there.

Convert Shared Libraries or Windows DLLs to Cilk++

1. You can create a Cilk++ Windows DLL; read the "***Converting Windows DLLs to Cilk++*** (Page 121)" chapter. Also, see the `qsort-dll` example.
2. For Linux shared objects, see the "***Linux Cilk++ Shared Library Creation*** (Page 32)" section.

Compile and Build Cilk++, Possibly on a Different Operating System

1. Read the "***Building Cilk++ for Linux*** (Page 129)" chapter.
2. Read the ***License*** (Page 137) terms.

Learn More about Cilk++ and its Implementation.

1. Read the "***Additional Resources and Information*** (Page 8)" section.
2. Visit the ***Cilk Arts, Inc.*** http://www.cilk.com web site.

Chapter 20

LICENSE

Some of the Cilk++ development tools may be used under the GPL open source license; others require a commercial license from Cilk Arts. The Cilk++ runtime, required for application deployment, may similarly be used under open source (CAPL) or commercial terms.

For more information about licensing Cilk Arts products, please visit the ***Cilk Arts web site*** http://www.cilk.com/home/try-and-buy-cilk/.

For your reference, the Cilk Arts Public License (CAPL) and the Cilk Arts commercial license terms are included in this document.

This work has been supported in part by DARPA contract W31P4Q-08-C-015.

This work has been supported in part by the NSF under grant IIP-0712243.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Government.

Some components of Cilk++ are subject to other license terms as acknowledged later in this chapter.

The most recent version of the Cilk Arts Public License is available at:

http://www.cilk.com/multicore-products/cilk-arts-public-license/
http://www.cilk.com/multicore-products/cilk-arts-public-license/

Version 1 is in the next section.

CILK ARTS PUBLIC LICENSE (CAPL)

Version 1, 5 November 2008

0. Definitions

The "Program" refers to any copyrightable work licensed under this License.

To "Modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "derivative work" of the earlier work. A work created by combining or linking the Program, in whole or in part, with other software is a derivative work.

The "Source Code" for a work means the human-readable form of the work that is optimal for making modifications to it, including all the source code needed to generate, install, and (for an executable work) run the object code and to Modify the work, including scripts to control those activities, but excluding Standard Interfaces and System Libraries for the work as defined in the Gnu General Public License ("GPL"), version 3.

"You" refers to the licensee of this program, and may be an individual or organization.

An "Individual Developer" means any human individual (not an organization) who contributes, in a nontrivial way, to the development or modification of the Source Code for a Program or a derivative work of a Program.

To "Share" a work is to provide costless public access to the work to everyone. The means of access must include making the work available via a web page that: a) is visible to all widely used internet search engines and all users of those search engines; and b) includes the Source Code and related documentation for the work and the phrase "This work is derived from a Program available under the terms of the Cilk Arts Public License".

1. Redistribution

Redistribution of the Program or derivative works of the Program, in source or binary forms, in standalone form or packaged with one or more applications that use the Program, is permitted provided that the following conditions are met:

- Redistribution in any form must be accompanied by information on how to obtain the complete Source Code and related documentation for the Program, and you must Share any derivative work of the Program.
- Each redistribution must bind all licensees to this list of conditions.
- Each redistribution must be pursuant to:
  - for your modifications, the terms of this license or a license of your choosing that is compatible with GPL (version 3 or earlier); and
  - for the portion of the derivative work which is the original Program, the terms of this license.
- Each redistribution must contain the following copyright notice and the following disclaimer in any Source Code, in any related documentation, and in any licenses for the redistribution.

PORTIONS OF THIS PROGRAM, INCLUDING THE CILK++ COMPILER AND RUNTIME SYSTEM, ARE PROVIDED BY CILK ARTS ``AS IS,'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, ARE DISCLAIMED. IN NO EVENT SHALL CILK ARTS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, PUNITIVE OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

2. Use of the program

You may use the Program if you accept the terms of this license and you acknowledge the copyright notice and disclaimer set forth in paragraph 1.

If you create a derivative work of the Program, any Individual Developers of your derivative work may use it if they accept the terms of this license.

You may also allow anyone who is not an Individual Developer of your derivative work to use it, but if you do so then: a) you must Share your derivative work with everyone for at least two years after the first such use and as long as you use, distribute, or support the derivative work; and b) allow it to be redistributed indefinitely under the conditions of paragraph 1 above. Notwithstanding the foregoing, you may, without being obliged to Share your derivative work or allow its redistribution, allow others to perform, on behalf of you and other Individual Developers of your derivative work, activities supporting development or evaluation of your derivative work, such as copying, backup, compiling, testing, and benchmarking.

CILK ARTS COMMERCIAL LICENSE

THE NUMBERED PARAGRAPHS BELOW SET FORTH THE CILK ARTS, INC. PROPRIETARY SOFTWARE INTERNAL USE LICENSE AGREEMENT (THE "LICENSE AGREEMENT"). WHEN ACKNOWLEDGED BY THE USER ACCESSING THIS SITE, THEY CONSTITUTE A LEGAL AGREEMENT BETWEEN CILK ARTS, INC. ("CILK ARTS") AND THE COMPANY OR INDIVIDUAL THAT IS THE END USER OF THE SOFTWARE (THE "CUSTOMER"). BY ACKNOWLEDGING ACCEPTANCE OF THIS LICENSE AGREEMENT, AND/OR BY INSTALLING AND USING THE ACCOMPANYING SOFTWARE (THE "SOFTWARE"), YOU REPRESENT THAT YOU HAVE BEEN AUTHORIZED TO ACCEPT AND ACKNOWLEDGE YOUR ACCEPTANCE OF THESE TERMS ON BEHALF OF THE CUSTOMER. IF YOU ARE NOT SO AUTHORIZED OR DO NOT AGREE TO BE BOUND BY THESE TERMS, IN PARTICULAR, THE LIMITATIONS ON USE IN SECTIONS A&B; WARRANTIES & LIABILITY IN SECTION C; AND TRANSERABILITY IN SECTION G, THEN CILK ARTS IS UNWILLING TO PERMIT ACCESS TO THE SOFTWARE OR THE INSTALLATION AND USE OF THE SOFTWARE, OR TO GRANT ANY LICENSE TO THE SOFTWARE, AND ANY INSTALLATION OR USE OF THE SOFTWARE WOULD BE A VIOLATION OF U.S. AND INTERNATIONAL COPYRIGHT LAWS. IF YOU ARE NOT WILLING TO ACCEPT THESE TERMS AND CONDITIONS, YOU MAY NOT ACCESS, INSTALL OR USE THE SOFTWARE.

PROPRIETARY SOFTWARE INTERNAL USE LICENSE AGREEMENT

If you have been provided with this Software pursuant to a separate written agreement signed by you and an authorized representative of Cilk Arts, such as the Cilk Arts Software License and Services Agreement, your use of the Software is governed by the terms and conditions of that agreement which will apply in place of the following. OTHERWISE YOU AGREE THAT THIS LICENSE AGREEMENT IS ENFORCEABLE TO THE SAME EXTENT AS ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

Section A: Types of License; Maintenance and Support

Your license requires purchase of the Software from Cilk Arts, Inc., either pursuant to a purchase order or by online payment in connection with download of the Software. The type of order or download will determine the scope of your license, as follows:

1. Development and Deployment License: If your purchase order or download is for a development license and deployment license, you are granted the personal, non-exclusive, non-transferable, limited license to use the Software to develop one or more Developed Applications in accordance with the terms of such purchase order or download and this License Agreement and to deploy such Developed Applications, each solely for use pursuant to a valid Runtime License from Cilk Arts.

2. Commercial Runtime License: If your purchase order or download includes one or more commercial Runtime Licenses, you are granted the right to use a Developed Application in accordance with the terms of such purchase order or download and this License Agreement.

3. Evaluation License: If your purchase order or download is for an evaluation license you are granted the personal, non-exclusive, non-transferable, limited license to use the Software (including to develop one or more Developed Applications) for a limited period of time for the purposes of evaluation for your individual use or internal business purposes only in accordance with the terms of this License Agreement. You agree to use commercially reasonable efforts to report results of your evaluation to Cilk Arts, including all errors discovered and a description of the environment in which the Software was evaluated. Cilk Arts may, without your permission, publish nonproprietary information received from you regarding your use of and comments about the Software.

4. Education License: If your purchase order or download is for an education license, you are granted the personal, non-exclusive, non-transferable, limited license to use the Software in perpetuity, solely for the internal purpose of supporting teaching faculty and students working within the curriculum of an accredited academic institution.

5. Maintenance & Support: If you purchase a Development and Deployment License or a commercial Runtime License, Cilk Arts will provide maintenance releases, updates and support for one year from the date of the purchase. Such maintenance release or updates shall be considered part of the Software and subject to the terms of this License Agreement.

Section B: License; Restrictions on License

1. All worldwide right, title and interest in and to the Software and any patents, copyrights, or other proprietary rights relating thereto, including with respect to any modifications, improvements or derivative works thereof, shall be and remain the exclusive property of Cilk Arts (and/or its licensors if applicable). Cilk Arts shall not acquire any right, title or interest in a Developed Application apart from its ownership of the Software and derivative works thereof.

2. You may not resell, distribute, rent, assign, sell, loan, lease, sublicense or use the Software to provide service bureau or similar services to third parties, nor deploy the Software for the benefit of another party. You may not disseminate or publish Software benchmarks of any kind without advance written consent of Cilk Arts.

3. You may copy the object code of the Software into a computer readable form for back-up purposes, solely in support of the licensed use of the Software. Except as specifically permitted in Section A.1 above, no other reproduction, copying, manufacture, modification, or creation of derivative works of the Software is permitted. You agree to affix, maintain and reproduce Cilk Arts' copyright, trademark, patent and proprietary rights notices, as may be applicable, on or in all Software and program media. You will not alter, remove, or otherwise disturb any such notices. A Developed Application shall not substantially replicate the functionality of the Software and must provide a function substantially different from the function of the Software.

4. USE OF THE SOFTWARE IS RESTRICTED TO THE COMPUTER SYSTEM(S); NUMBER OF CPUS, USERS, SEATS, PROCESSING CHIPS, DATA COMMUNICATION SOCKETS, SERVERS OR THE LIKE; PERIOD OF TIME; AND PURPOSE (E.G., DEVELOPMENT, DEPLOYMENT, RUNTIME USE, EVALUATION, OR EDUCATIONAL) AS LICENSED BY CILK ARTS AND FOR WHICH ANY FEES DUE HAVE BEEN PAID. In the event that any such parameter is exceeded, the license is not valid and the Software may not function properly or at all. Accordingly, in such circumstances, use of the Software and access to any data, files or output created with the Software or any product associated with the Software is entirely at your own risk. In the absence of any specified time, evaluation licenses remain valid for a period of 90 days only from the date of acceptance of this License Agreement. At the expiration of the evaluation period you must destroy all originals and copies of evaluation Software. In the event that you fail to destroy any evaluation Software by such time, payment will immediately become due for the full license price and cost of support and maintenance (supplied by Cilk Arts at its sole option) for an initial period of 12 months for each license as stated in the Price List current at the time of invoice, plus VAT (if applicable) and any other relevant taxes. You will defend, indemnify and hold Cilk Arts and its licensors harmless from any claims (including reasonable attorneys' fees) based on your use of the Software other than as permitted under this License Agreement.

5. You must maintain complete and accurate records indicating the location where each copy of the Software has been installed, and as applicable, the Computer System and the number of CPUs/Users/Seats/Servers for each copy.

6. Except as expressly permitted by law, you shall not reverse engineer, decompile, decode, or disassemble or derive the source code of the Software.

7. (a) You acknowledge that the Software, including technical data, is subject to export controls under the laws of the United States and other jurisdictions from which they are exported or re-exported. You shall be solely responsible for compliance with such applicable export laws and you shall not export, re-export or use any of the Software in violation of export control laws and regulations. (b) You also shall not export and/or re-export Software, including technical data, to individuals or companies listed on the U.S. Department of Commerce's Denied Parties List and the Entity List, on the U.S. Department of Treasury's Specially Designated Nationals Lists, or any other list of parties proscribed by the U.S Government. (c) You shall not export and/or re-export Software and technical data, if you know or have reason to know that the end user is engaged in the design, development and use of nuclear, chemical and biological, and/or missile technology activities. (d) Software acquired with United States Federal Government funds or intended for use within or for any United States federal agency are provided solely pursuant to the terms of this License Agreement and with "Restricted Rights" as defined in DFARS 252.227-7013(c)(1)(ii) or FAR 52.227-19.

8. If Cilk Arts reasonably suspects that you have breached this License Agreement, Cilk Arts may audit your Software-related activities upon 24 hours' notice.

9. All Software is provided © 2007-2009. Cilk Arts, Inc. Patents pending. All rights not expressly granted are reserved to Cilk Arts.

10. You acknowledge that in certain cases the Software may be provided for operation in cooperation with certain Open Source Components, the use of which shall be governed by the terms of an applicable open source license agreement. If the Software is supplied with Open Source Components or third party products or services, it may only be used with such items.

11. If having purchased a valid Development and Deployment license you deploy one or more Developed Applications to one or more users (whether internal or external to your business), you shall: (a) restrict the use of such Developed Applications to users who have valid Runtime Licenses; (b) pass through the terms of Section C of this License Agreement to any such users; (c) bind your users to such terms and conditions as may be required with respect to any other programs or components supplied together with the Software, including, without limitation, any Open Source Components; and (d) reproduce in all Developed Applications any legends set forth in the Software.

Section C: Warranties and Limitations on Liability

1. Development and Deployment Licenses; Commercial Runtime Licenses: Solely in respect of such licenses described in Sections A1 and A2:

1.1. Cilk Arts warrants that to its knowledge the use of unmodified Software in accordance with this License Agreement will not violate the rights of any third party under United States copyright, trademark or trade secret law. In the case of any alleged breach of this section C1 by Cilk Arts, Cilk Arts shall, at its expense, indemnify, defend, and hold harmless you from and against any loss, cost, damage or expense arising from any third party claim provided that (a) you promptly give Cilk Arts written notice of such claim and provide all reasonable assistance and (b) you give Cilk Arts the right to control the defense or settlement of the claim with counsel of its own choosing.

1.2. Cilk Arts warrants that for the first 90 days after installation of the Software ("Warranty Period"), the unmodified Software will substantially conform to the User Documentation. Your sole and exclusive remedy for any breach by Cilk Arts of this warranty shall be that during the Warranty Period, Cilk Arts shall, in its sole discretion, modify the relevant Software to bring it into substantial conformance with the User Documentation, replace the Software, or refund the license fees paid by you for the defective Software.

2. Evaluation Licenses: With respect to Software obtained pursuant to an evaluation license:

2.1. You acknowledge that Software provided for evaluation is experimental and does not represent final product from Cilk Arts and may have bugs, errors defects or deficiencies which cannot or will not be corrected by Cilk Arts and which could cause system or other failures and data loss.

2.2. You acknowledge that Cilk Arts has no obligation to continue to provide any Software provided under an evaluation license.

2.3. You acknowledge that evaluation Software is provided solely for you to satisfy yourself that such Software performs as required in your operating environment and to your specific needs.

2.4. Evaluation Software will terminate operation after a designated period of time following installation as defined within the Software. Upon such date the license hereunder shall be terminated and such Software shall cease operation. Accordingly, access to any files or output created with such Software or any product associated with the Software is entirely at your own risk.

2.5. ACCORDINGLY YOU ARE ADVISED NOT TO RELY ON SUCH EVALUATION SOFTWARE FOR ANY REASON AS IT IS PROVIDED "AS IS," WITH NO EXPRESS OR IMPLIED CONDITION OR WARRANTY WHATSOEVER. TO THE EXTENT LIABILITY CANNOT BE EXCLUDED, BUT MAY BE LIMITED, CILK ARTS'S LIABILITY SHALL BE LIMITED TO THE SUM OF FIFTY DOLLARS (U.S. $50.00) IN TOTAL.

3. Education Licenses: With respect to Software obtained pursuant to an education license:

You acknowledge that Software is provided solely for solely for supporting the curriculum of an accredited academic institution. ACCORDINGLY YOU ARE ADVISED NOT TO RELY ON SUCH SOFTWARE FOR ANY REASON AS IT IS PROVIDED "AS IS," WITH NO EXPRESS OR IMPLIED CONDITION OR WARRANTY WHATSOEVER. TO THE EXTENT LIABILITY CANNOT BE EXCLUDED, BUT MAY BE LIMITED, CILK ARTS'S LIABILITY SHALL BE LIMITED TO THE SUM OF FIFTY DOLLARS (U.S. $50.00) IN TOTAL.

4.General:

4.1. Some jurisdictions may grant you additional rights and/or do not allow the exclusion of or limitation on implied conditions, warranties or damages, so the following shall apply to the extent permitted by law applicable to you in your jurisdiction.

4.2. Except for the express limited warranties set forth in SECTION C1, Cilk Arts and its licensors make NO WARRANTY OF ANY KIND WITH RESPECT TO ANY SOFTWARE PROVIDED HEREUNDER AND HEREBY EXPRESSLY EXCLUDE ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE INCLUDING, WITHOUT LIMITATION ANY WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE. CILK ARTS AND ITS LICENSORS MAKE NO WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, IN RELATION TO: (a) THE MEDIA OF ANY COPIES OF SOFTWARE MADE BY YOU; (b) ANY DERIVATIVE WORK, APPLICATION OR SOLUTION MADE USING THE SOFTWARE; OR (c) ANY MODIFIED SOFTWARE; OR (d) ANY SOFTWARE NOT USED IN COMPLIANCE WITH THIS LICENSE AGREEMENT AND THE USER DOCUMENTATION. Cilk Arts does not warrant that the use of Software will be uninterrupted or error free or that all problems or errors will be remedied.

4.3. IN NO EVENT WILL CILK ARTS OR ITS LICENSORS BE LIABLE FOR ANY PUNITIVE, SPECIAL, INDIRECT, CONSEQUENTIAL OR INCIDENTAL LOSSES OR DAMAGES, WHETHER FORESEEABLE OR NOT, OR LOST REVENUES, GOODWILL OR PROFITS, BUSINESS INTERRUPTION, PROCUREMENT OF SUBSTITUTE SOFTWARE OR SERVICES, LOST DATA, WORK STOPPAGE OR DELAYS, RE-RUN TIME, INACCURATE OUTPUT, COMPUTER FAILURE OR MALFUNCTION.

4.4. CILK ARTS' AND ITS LICENSORS' LIABILITY TO YOU OR ANY OTHER PARTY FOR A CLAIM OR SERIES OF RELATED CLAIMS OF ANY KIND RELATED TO THIS LICENSE AGREEMENT OR ANY SOFTWARE OR SERVICE, WHETHER FOR BREACH OF CONTRACT OR WARRANTY, STRICT LIABILITY, NEGLIGENCE OR OTHERWISE, SHALL BE LIMITED TO DIRECT LOSSES AND MONETARY DAMAGES ONLY AND SHALL NOT EXCEED THE AGGREGATE AMOUNT PAID TO CILK ARTS BY YOU FOR THE SOFTWARE TO WHICH THE CLAIM RELATES.

4.5. NO ACTION, REGARDLESS OF FORM, ARISING OUT OF THE TRANSACTION UNDER THIS LICENSE AGREEMENT MAY BE BROUGHT BY YOU MORE THAN ONE YEAR AFTER THE EVENTS WHICH GAVE RISE TO THE CAUSE OF ACTION OCCURRED.

4.6. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY REMEDY PROVIDED HEREIN.

4.7. Cilk Arts is acting on behalf of its licensors and suppliers for the purpose of disclaiming, excluding and/or limiting obligations, warranties and liability as provided in this License Agreement, but in no other respects and for no other purpose.

Section D: Charges and Payment

1. Unless otherwise agreed upon by Cilk Arts in writing, you shall pay the current Cilk Arts list price for licenses for the Software. All charges stated are exclusive of taxes (including value added tax), tariffs and transportation expenses which shall be payable in addition thereto. Cilk Arts reserves the right to charge interest on overdue amounts at the lesser of 1.5% per month or the then-current statutory maximum interest rate, from the date such amounts become due to the date on which payment is credited to Cilk Arts's account.

2. Without prejudice to your rights, if any, against Cilk Arts, all charges shall be paid without deductions of any amount, whether in the nature of offset, counterclaim or otherwise.

Section E: Termination

1. This License Agreement will terminate: (a) upon expiration of the copyrights in the Software or at Cilk Arts's option, if any Software becomes, or in Cilk Arts opinion, is likely to become, the subject of a claim of infringement of a patent, trade secret or copyright; or (b) upon written notice from Cilk Arts to you for any material breach of this License Agreement; or (c) if you are subject to any proceedings relating to insolvency or bankruptcy or any similar proceedings under applicable law; or (d) upon termination by you by written notice to Cilk Arts.

2. Upon termination for any reason: (a) all of your rights will terminate immediately and all copies of the Software shall be destroyed; and (b) all amounts due to Cilk Arts hereunder will become immediately due and payable. There shall be no refunds of any sums already paid and any provisions which are expressly or by implication intended to survive termination will so survive.

Section F: Confidentiality

1. You shall not use or disclose any Confidential Information of Cilk Arts except as expressly permitted by this Agreement. You shall use the highest commercially reasonable degree of care to protect the Confidential Information. The foregoing does not apply to information: (a) rightfully known prior to receipt; (b) that becomes public knowledge by acts other than by you after receiving such information; (c) that is independently developed by you without a breach of obligations hereunder; or (d) that is rightfully received by you from a third party without restriction and without breach of this License Agreement.

2. Nothing in this Section F shall prevent you from disclosing all or part of the Confidential Information which is necessary to disclose pursuant to the lawful requirement of a governmental agency or when disclosure is required by operation of law, rule or regulation, provided, however, that prior to any such disclosure, you shall: (a) promptly notify Cilk Arts in writing of such requirement to disclose; and (b) co-operate fully with Cilk Arts in protecting against any such disclosure and/or obtaining a protective order.

Section G: Miscellaneous

1. You may not assign this License Agreement without the prior written consent of Cilk Arts, including in connection with a sale of stock or assets, merger or similar transaction. Subject to the foregoing, this License Agreement shall bind and inure to the benefit of any successors and assigns. Cilk Arts reserves the right to assign this License at any time. No relationship of employment, co-development, independent contracting or otherwise is created under this License Agreement.

2. Neither party will be responsible for delays in or failure of performance due to causes beyond its reasonable control.

3. Together with your purchase order or download this License Agreement is the complete and exclusive statement of the parties in relation to the subject matter hereof; it sets forth all obligations of Cilk Arts to you in relation to the subject matter hereof; it supersedes all prior or simultaneous written or oral proposals and understandings relating thereto, all of which are expressly excluded; it takes precedence over the terms of any purchase order/written request issued by you (including any pre-printed information included in your purchase order/written request); and it can only be modified by a written amendment signed by authorized representatives of both parties.

4. If any court of competent jurisdiction determines that any provision or part of this License Agreement is invalid, the remainder will continue in full force and effect and the offending provision or part shall be interpreted to whatever extent possible to give effect to its stated intent.

5. . This License Agreement is governed by the laws of Massachusetts, without giving effect to its conflict of law provisions, and the parties hereby irrevocably submit themselves and waive any and all defenses to the exclusive jurisdiction of the courts of Massachusetts for any controversy arising out of this License Agreement. The United Nations Convention on Contracts for the International Sale of Goods will not apply to the transactions under this License Agreement.

Section H: Definitions

1. "Computer System" means a combination of computer hardware (consisting of one or more Servers) and operating system(s) or, if partitioned, each logical partition thereof, in which any portion of the Software is installed.

2. "Confidential Information" means (a) pricing or information concerning the Software and any other Cilk Arts products or services; (b) Cilk Arts trade secrets and other proprietary information (including technical information and any source code or binary code of the Software and any pre-commercial release Software); and (c) any business, marketing or technical information disclosed by Cilk Arts and identified in writing as confidential or proprietary to the disclosing party.

3. "CPU" means all central processing unit processors resident in a Computer System.

4. "Developed Application" shall mean an application developed using the Software either: (a) under a valid development license such as described in Section A.1 of this License Agreement; or (b) pursuant to another valid license agreement from Cilk Arts, such as the Cilk Arts Public License.

5. "Open Source Components" shall mean one or more software modules provided by Cilk Arts or its licensors that operate in cooperation with the Software and that are provided on open source licensing terms separate from this License Agreement. Nothing in this Agreement is intended to negate such open source licensing terms.

6. "Runtime License" shall mean a license from Cilk Arts for a single user to use a single Developed Application on a single processor chip via a single data communication socket and may include any of: (a) a commercial runtime license purchased pursuant to the terms of this License Agreement or upon similar terms purchased through a reseller, distributor, or sub-licensee of Cilk Arts; or (b) a runtime license obtained pursuant to a valid open source license from Cilk Arts, such as a current version of the Cilk Arts Public License.

7. "Server" means a single identified physical Computer System and an identified number of CPUs licensed for such Server.

8. "User," for Software licensed on a per user basis, is defined as the maximum number of connections to Software at any one instance in time; therefore, when an individual has multiple connections, each connection is a User. A multiplexing front end, such as a transaction manager, does not reduce the number of Users; the User count equals the number of connections from the clients to the multiplexing front end, rather than the smaller number of connections from the multiplexing front end to a database.

9. "User Documentation" means the standard documentation made available by Cilk Arts with respect to the Software that describes the functional and other specifications of, and methods of use for the Software, as the same may be amended from time to time by Cilk Arts.

GCC

The GCC components are distributed under the GNU General Public License; a copy of the GPL is included in the Cilk++ source distribution, and may also be viewed at:

http://www.gnu.org/copyleft/gpl.html http://www.gnu.org/copyleft/gpl.html

PIN

Cilkscreen is built using technology called *Pin* produced by Intel and distributed under the following license:

Intel Open Source License Copyright (c) 2003 Intel Corporation
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the Intel Corporation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

OCAML

Some components of the system are built using the *OCAML* compiler, linked with the *OCAML Library*, and distributed under the terms of the following license:

The Library is distributed under the terms of the ***GNU Library General Public License version 2*** http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html. As a special exception to the GNU Library General Public License, you may link, statically or dynamically, a "work that uses the Library" with a publicly distributed version of the Library to produce an executable file containing portions of the Library, and distribute that executable file under terms of your choice, without any of the additional requirements listed in clause 6 of the GNU Library General Public License. By "a publicly distributed version of the Library", we mean either the unmodified Library as distributed by INRIA, or a modified version of the Library that is distributed under the conditions defined in clause 3 of the GNU Library General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU Library General Public License.

WIX

Cilk Arts appreciates the efforts of the authors of the *WIX* installation toolkit for providing tools used to build the installation packages for Microsoft Windows. For more information about *WIX*, visit the ***WIX project on SourceForge*** (http://sourceforge.net/projects/wix/).

YASM

Cilk Arts appreciates the efforts of the authors of the open source assembler *Yasm*. For more information about *Yasm*, visit the home page of the ***Yasm Modular Assembler*** (http://www.tortall.net/projects/yasm/wiki) project.

Yasm is Copyright (c) 2001-2008 Peter Johnson and other Yasm developers.

Yasm developers and/or contributors include:

- Peter Johnson
- Michael Urman
- Brian Gladman (VC8 build files, other fixes)
- Stanislav Karchebny (options parser)
- Mathieu Monnier (SSE4 instruction patches)
- Anonymous "NASM64" developer (NASM preprocessor fixes)
- Stephen Polkowski (x86 instruction patches)
- Henryk Richter (Mach-O object format)
- Ben Skeggs (patches, bug reports)

Yasm, along with modifications made by Cilk Arts, is redistributed according to the terms of the BSD licenses referenced below:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the author nor the names of other contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND OTHER CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR OTHER CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ZEDGRAPH

The ZedGraph DLL is distributed under the GNU Lesser General Public License. The license may be viewed ***here*** http://www.gnu.org/copyleft/lesser.html.

The source code for ZedGraph is in the Cilk++ for Windows installation; the default location and file names are:

```
c:\Program Files\Cilk Arts\Cilk++\opensrc\zedgraph_source_v515.zip
```

The file, `zedgraph-readme.txt`, is in the same directory.

GLOSSARY OF TERMS

A

About the Glossary

The Glossary is an alphabetical list of important terms used in the Cilk++ 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. See the Cilk Arts home page for more information.

cilk_for

A keyword in ***Cilk++*** that indicates a `for` loop whose iterations can be executed independently in parallel.

cilk_spawn

A keyword in ***Cilk++*** that indicates that the named subroutine can execute independently and in parallel with the caller.

cilk_sync

A keyword in ***Cilk++*** 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. Cilk++ is based on the Cilk programming language which ***Cilk Arts, Inc.*** licensed from MIT in order to commercialize the technology.

Cilkscreen

The ***Cilkscreen Race Detector*** tool provided by ***Cilk Arts, Inc.***, 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 the ***Cilkscreen Race Detector*** 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 in ***Cilk++*** 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 that Cilk++ currently supports.

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++ execution 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 supports several mutually exclusive locks, including the `CRITICAL_SECTION`. Linux 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++*** keyword ***cilk_for*** 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:

1. A default constructor which initializes the reducer to its identity value
2. 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 Cilk Arts blog "***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***. ***Cilk++*** employs a work-stealing scheduler.

worker

A ***thread*** that, together with other workers, implements the ***Cilk++*** runtime system's ***work stealing*** scheduler.

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z