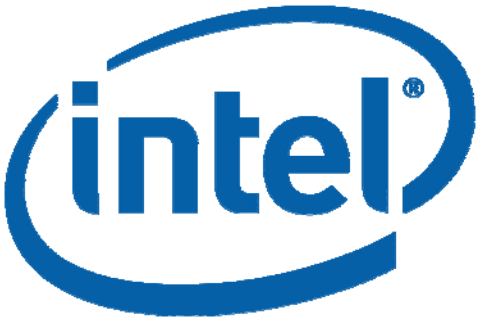


Cilk User's Guide

Document Number: 322581-001US

Legal Information

Contents

1 Introduction

This document describes use of Cilk technology with the Intel® C++ Compiler. This technology allows you to add parallelism to new or existing C or C++ programs. This release provides support for building IA-32 and Intel® 64 architecture programs (32-bit and 64-bit) that run on the Microsoft Windows* operating system (OS) and IA-32 and Intel® 64 architecture programs (32-bit and 64-bit) that run on the Linux* operating system.

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

1.1 Target Audience

This document is designed for application developers who will use Cilk to improve performance by adding parallelism to new and existing C or C+ applications. Application developers need at least a working knowledge of C or C++ programming: expert-level knowledge of C or C++ is optimal.

1.2 Prerequisites

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

Install Intel® Parallel Composer 2011 or Intel® C++ Composer XE 2011, then build and run at least one of the Compiler sample programs in order to validate the installation and to begin to get familiar with the compiler and tools.

1.3 Typographic Conventions

A `monospaced font` denotes commands, code, keywords and program output.

An *italicized font* indicates a word that is defined in the Glossary of Terms.

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

1.4 Additional Resources and Information

Appendix B: Reading List contains a list of additional resources.

2 Getting Started

Cilk adds fine-grained task support to C and C++, making it easy to add parallelism to both new and existing software to 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 (tasks) that can be solved independently, then combining the results. Recursive functions are often used for divide and conquer algorithms, and are well supported by Cilk.

Tasks can either be implemented in separate functions or in iterations of a loop. The Cilk keywords identify function calls and loops that can run in parallel. The Cilk runtime schedules these tasks to run efficiently on the available processors.

This chapter covers the following:

- Building, running and testing a sample Cilk program.
- Converting a simple C++ program to use Cilk.
- Building the converted program, testing it for race conditions, and measuring its parallel performance.

In the following sections, the term ***worker*** is used to mean an operating system thread used to execute a task in a Cilk program.

2.1 Build and Run a Cilk Example

Each example resides in an individual folder, as described in the Release Notes. Use of the `qsort` example will be described here.

You should have already installed the product (Intel® Parallel Composer 2011 or Intel® C++ Composer XE 2011) on a system that has more than one processor core available. If you have a single-core system, you can build and test the example, but you should not expect to see any performance improvements.

2.1.1 Building qsort

Full, detailed build options are described in the Building, Running, and Debugging chapter. For now, use the default settings.

The Intel® C++ compiler command name is `icl` on Windows* systems and `icc` on Linux* systems.

Linux* Systems

1. Change to the qsort directory (`cd INSTALLDIR/examples/qsort`)
2. Issue the `make` command. If `make` fails, check to be sure that the PATH environment variable is set to find Cilk from `INSTALLDIR/bin`.
3. The executable `qsort` will be built in the current directory.

Windows* Systems

Microsoft Visual Studio* users: Open the solution (such as `qsort.sln`) and build the `Release` version. The executable will be built as `EXDIR\qsort\Release\qsort.exe`.

From the command line, build qsort.exe using the icl command: `icl qsort.cpp`

If you get an error message, verify that the environment is set up to use the Intel® C++ Compiler. Right-click on the project name and, depending on your installed product, select one of the following:

- **Intel Parallel Composer 2011>Use Intel C++**
- **Intel C++ Composer XE 2011>Use Intel C++**

2.1.2 Running qsort

First, ensure that `qsort` runs correctly. With no arguments, the program will create and sort an array of 10,000,000 integers. In the following, the > represents the command prompt:

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

2.1.3 Observe speed up on a multicore system

By default, a Cilk program queries the operating system and uses as many cores as it finds. You can control the number of workers using the `CILK_NWORKERS` environment variable.

The following results occur on an 8-core system, where the speed up is limited by the application's parallelism and the core count.

Linux* OS:

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

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

>CILK_NWORKERS=4 qsort
Sorting 10000000 integers
0.798 seconds
Sort succeeded.

>CILK_NWORKERS=8 qsort
Sorting 10000000 integers
0.438 seconds
Sort succeeded.
```

Windows* Command Line:

```
>set CILK_NWORKERS=1
>qsort
Sorting 10000000 integers
2.909 seconds
Sort succeeded.

>set CILK_NWORKERS=2
>qsort
Sorting 10000000 integers
1.468 seconds
Sort succeeded.

>set CILK_NWORKERS=4
>qsort
Sorting 10000000 integers
0.798 seconds
Sort succeeded.

>set CILK_NWORKERS=8
>qsort
Sorting 10000000 integers
0.438 seconds
Sort succeeded.
```

Microsoft Visual Studio*:

Right-click on the project and select **Properties** from the context menu.

In the **Configuration** category, choose **Intel Debugging** and use the **Number of Cilk Threads** property to specify the number of workers.

2.2 Convert a C++ Program

The sequence of steps to create a parallel program using Cilk is as follows:

1. Typically, you start with a serial C++ program that implements the basic functions or algorithms that you want to parallelize. Ensure that the serial program is correct. Any bugs in the serial program will occur in the parallel program, but they will be more difficult to identify and fix.
2. 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.
3. Use the three Cilk keywords to identify tasks that can execute in parallel:
   - `_Cilk_spawn` (or, `cilk_spawn`, if your program includes `<cilk/cilk.h>`) indicates a call to a function (a "child") that can proceed in parallel with the caller (the "parent").
   - `_Cilk_sync` (or, `cilk_sync`, if your program includes `<cilk/cilk.h>`) indicates that all spawned children must complete before proceeding.
   - `_Cilk_for` (or, `cilk_for`, if your program includes `<cilk/cilk.h>`) identifies a loop for which all iterations can execute in parallel.
4. Build the program:
   - **Windows* OS:** Use either the `icl` command-line tool or compile within Microsoft Visual Studio*. If using Visual Studio*, make sure that you have selected **Use Intel C++** from the product context menu.
   - **Linux* OS:** Use the `icc` command.
5. Run the program. If there are no *race conditions*, the parallel program produces the same result as the serial program.
6. Correct any race conditions with *reducers*, locks, or recode to resolve conflicts.

A walk-through of this process follows, using a sort program as an example.

2.2.1 Start with a Serial Program

The following demonstrates how to use write a Cilk program by parallelizing a simple implementation of Quicksort.

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

```
 9  #include <algorithm>
10
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.
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));
28          std::swap(*end, *middle);      // 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
45      sample_qsort(a, a + n);

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  }
```

2.2.2 Add Parallelism using _Cilk_spawn

You are now ready to introduce parallelism into the `qsort` program.

The `_Cilk_spawn` keyword indicates that a function (the *child*) may be executed in parallel with the code that follows the `_Cilk_spawn` statement (the *parent*). The keyword allows but does not require parallel operation. Cilk dynamically determines which operations are executed in parallel when multiple processors are available. The `_Cilk_sync` statement indicates that the function may not continue until all preceding `_Cilk_spawn` requests in the same function have completed. `_Cilk_sync` does not affect parallel strands spawned in other functions.

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

The previous example can be rewritten to use a simpler form of the Cilk keywords. Include the header file `<cilk/cilk.h>`, which defines macros that provide names in lowercase, without the initial underscore. The following shows the simpler naming convention of `cilk_spawn` and `cilk_sync`. This naming convention is used throughout the rest of the book.

```
19 #include <cilk/cilk.h>
21  void sample_qsort(int * begin, int * end)
22  {
23      if (begin != end) {
24          --end;  // Exclude last element (pivot)
```

```
25          int * middle = std::partition(begin, end,
26                        std::bind2nd(std::less<int>(),*end));
28          std::swap(*end, *middle);      // pivot to middle
29          cilk_spawn sample_qsort(begin, middle);
30          sample_qsort(++middle, ++end); // Exclude pivot
31          cilk_sync;
32      }
33  }
```

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

There is an implicit `cilk_sync` at the end of every function that waits until all tasks spawned in the function have returned, so the `cilk_sync` at line 31 is redundant, but included here for clarity.

The above change implements a typical divide and conquer strategy for parallelizing recursive algorithms. At each level of recursion, two-way parallelism occurs; the parent strand (line 29) continues executing the current function, while a child strand executes the other recursive call. This recursion can expose quite a lot of parallelism.

2.2.3 Build, Execute and Test

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

Linux* OS:

```
icc qsort.cpp -o qsort
```

Windows* Command Line:

```
icl qsort.cpp
```

Windows Visual Studio*:

Build the `Release` configuration.

2.2.3.1 Run qsort from the command line

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

2.2.3.2 Observe speed up on a multicore system

By default, a Cilk program queries the operating system and use all available cores. You can control the number of workers by setting the `CILK_NWORKERS` environment variable.

Run `qsort` using one and then two cores. On a system with two or more cores, you should expect to see that the second run takes about half as long as the first run.

Linux* OS:

```
>CILK_NWORKERS=1 qsort
Sorting 10000000 integers
Sort succeeded.

>CILK_NWORKERS=2 qsort
Sorting 10000000 integers
Sort succeeded.
```

Windows* Command Line:

```
>set CILK_NWORKERS=1
>qsort
Sorting 10000000 integers
Sort succeeded.

>set CILK_NWORKERS=2
>qsort
Sorting 10000000 integers
Sort succeeded.
```

3 Build, Run, and Debug a Cilk Program

This chapter shows how to build, run, and debug Cilk programs.

3.1 Set Worker Count

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

You can increase or decrease the number of workers under program control or via the environment. You may want to use fewer workers than the number of processor cores available in order to run tests or to reserve resources for other programs. In some cases, you may want to oversubscribe by creating more workers than the number of available processor cores. This may be useful if you have workers waiting on locks, or if you want to test a parallel program on a single-core computer.

3.1.1 Environment

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

Windows* OS: `set CILK_NWORKERS=4`

Linux* OS: `export CILK_NWORKERS=4`

3.1.2 Program Control

Prior to the first call to a function that performs a spawn (`cilk_spawn` or `cilk_for`) within a program, you may set the desired number of workers by calling `__cilkrts_set_param("nworkers","N")`, where *N* is a number expressed in decimal or, with a leading `0x` or `0`, in hexadecimal or octal. This call to set the worker count overrides the value set in the `CILK_NWORKERS` environment variable. To use `__cilkrts_set_param`, make sure to specify `#include <cilk/cilk_api.h>` in your program.

Example:

```
if (0!= __cilkrts_set_param("nworkers","4"))
{
```

```
    printf("Failed to set worker count\n");
    return 1;
}
```

3.2 Serialization

The Cilk keywords are designed to have serial semantics. In other words, every Cilk program corresponds to an equivalent C/C++ program. Such a C/C++ program is referred to as the ***serialization*** of the Cilk program. The serialization is particularly useful when debugging Cilk programs.

3.2.1 How to Create a Serialization

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

Linux* OS:

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

```
-cilk-serialize
-include cilk/cilk_stub.h
```

For example, to build the serialized `reducer` example, build `reducer.cpp` (which contains Cilk keywords) as follows:

```
icc -O2 –Qcilk-serialize -o reducer_serial reducer.cpp
```

Windows* OS:

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

```
/Qcilk-serialize
/FI cilk/cilk_stub.h
```

For example, to build the serialized `reducer` example, build `reducer.cpp` (which contains Cilk keywords) as follows:

```
icl /O2 /Qcilk-serialize reducer.cpp
```

3.3 Debugging Strategies

Debugging a parallel program tends to be more difficult than debugging a serial program. Use of Cilk is designed to simplify the challenge of parallel debugging as much as possible. Start by debugging the serialization first.

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

- If you are converting an existing C/C++ program, debug and test the serial version first.
- Once you have a parallel Cilk program, test and debug the serialization. Because both the serial base program and the serialization of the Cilk program are serial C/C++ programs, you can use existing serial debug tools and techniques.
- You can use the standard debuggers (gdb for Linux* and the Visual Studio* debugger for Windows), which have been enhanced to work with Cilk programs, although the results may sometimes be harder to interpret.
- If your program works correctly as a serialization or when run with only one Cilk worker, but behaves incorrectly when run with multiple Cilk workers, you probably have a data race. If this is the case, do one or more of the following:
  - Restructure the code to remove the race
  - Use a Cilk reducer
  - Use a mutual-exclusion lock (such as one of the mutex locks available as part of Intel® Threading Building Blocks), other lock, or atomic operation

It may be simpler to debug programs built without optimizations. Debugging without optimizations turns off inlining, resulting in a more accurate call stack; additionally, the compiler does not attempt to reorder instructions and optimize register usage.

4 Summary of Cilk Language Features

There are a number of Cilk-related elements such as keywords, command line options, environment variables and pragmas. The following table provides a summary.

Cilk Element	Description
Keywords	
_Cilk_spawn	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
_Cilk_sync	Indicates that the current function cannot continue past this point in parallel with its spawned children
_Cilk_for	Specifies a loop that permits loop iterations to run in parallel; is a replacement for the normal C/C++ `for` loop. This statement divides a loop into chunks containing one or more loop iterations. Each chunk is executed serially, and is spawned as a chunk during the execution of the loop.
Pragma:	
cilk grainsize	Specifies the grain size for one `cilk_for` loop
Predefined Macro:	
__cilk	Specifies the Cilk version number. The value is set at 200 for this release.
Environment Variable:	
CILK_NWORKERS	Specifies the number of worker threads.
Compiler Options:	
/Qcilk-serialize, cilk-serialize	Specifies that a cilk program should be

	serialized.
/Qintel-extensions[-], -[no]intel-extensions	Enables or disables Intel language extensions (which include Cilk). Intel language extensions are on by default.
Header files (located in include/cilk):	
cilk.h	Defines macros that provide names with simpler conventions (`cilk_spawn`, `cilk_sync` and `cilk_for`)
cilk_api.h	Declares the Cilk runtime functions and classes
cilk_stub.h	Can be used to serialize a Cilk application
reducer.h	Contains common reducer definitions.
reducer_list.h reducer_max.h reducer_min.h reducer_opadd.h reducer_opand.h reducer_opor.h reducer_opxor.h reducer_ostream.h reducer_string.h	Reducers provided in the reducer library. See Reducer Library.

5 The Cilk Keywords

This chapter describes the three Cilk keywords: `_Cilk_spawn`, `_Cilk_sync` and `_Cilk_for`.

The header file `<cilk/cilk.h>` defines macros that provide names with simpler conventions (`cilk_spawn`, `cilk_sync` and `cilk_for`). This chapter uses the names as defined in `cilk.h`.

5.1 cilk_spawn

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

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

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

cilk_spawn func(args);      // func() may return void
```

func is the name of a function which may run in parallel with the current ***strand***. A strand is a serial sequence of instructions without any parallel control. The execution of the routine containing the `cilk_spawn` can execute in parallel with the portion of *func* after the routine.

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

args are the arguments to the function being spawned. These arguments are evaluated before the spawn takes effect. 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. This is an example of a ***data race***.

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.

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);
```

You cannot spawn a function as an argument to another function:

```
g(cilk_spawn f());  // Not allowed
```

The correct approach is to spawn a function that calls both f() and g(). This is easily accomplished using a C++ lambda:

```
cilk_spawn [&]{ g(f()); }();
```

Note that the above is different from the following:

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

In the latter statement, `f()` is executed in the parent before spawning `g()` whereas with the lambda, `f()` and `g()` are both executed in the child.

5.2 cilk_sync

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

The syntax is as follows:

```
cilk_sync;
```

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

There is an implicit `cilk_sync` at the end of every function and every try block that contains a `cilk_spawn`. The Cilk semantic is defined this way for these 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.

5.3 cilk_for

A `cilk_for` loop is a replacement for the normal C/C++ `for` loop that permits loop iterations to run in parallel.

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 within 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.

The Cilk runtime converts a `cilk_for` loop into an efficient divide-and-conquer recursive traversal over the loop iterations.

Sample `cilk_for` loops include:

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

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

In C, but not C++, the loop control variable can be declared in advance:

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

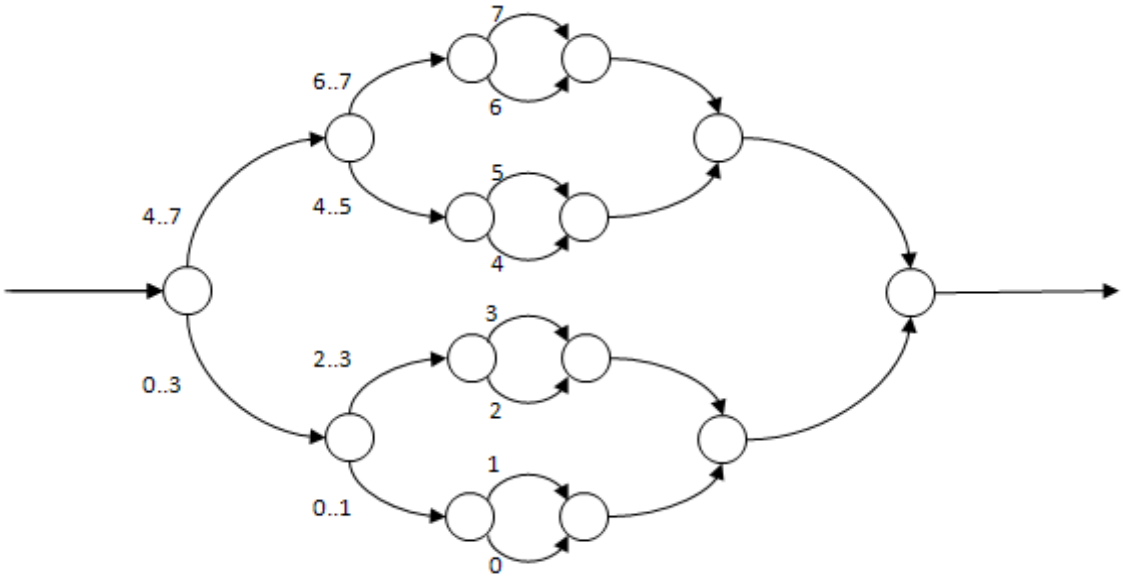
The serialization of a valid Cilk program has the same behavior as the similar C/C++ program, where the serialization of `cilk_for` is the result of replacing `cilk_for` with `for`. Therefore, a `cilk_for` loop must be a valid C/C++ `for` loop, but `cilk_for` loops have several constraints compared to C/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, as that would cause a *data race*. A reducer can often be used to prevent a race.

5.3.1 Serial/parallel structure of cilk_for

Using `cilk_for` is *not* the same as spawning each loop iteration. In fact, the Intel compiler converts the loop body to a function that is called recursively using a divide-and-conquer strategy; this provides significantly better performance. The difference can be seen clearly in the Directed Acyclic Graph (DAG) for the two strategies.

First, consider the DAG for a `cilk_for`, assuming N=8 iterations and a grain size of 1. The numbers label the serial sequence of instructions, known as ***strands***; these numbers indicate which loop iteration is handled by each strand.



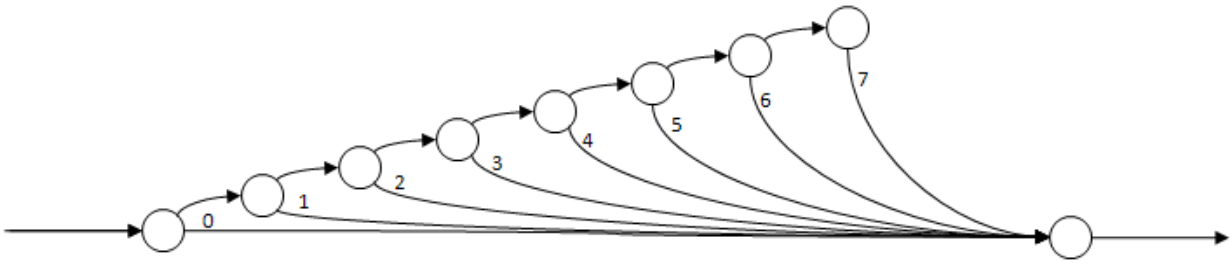
At each division of work, half of the remaining work is done in the child and half in the continuation. Importantly, the overhead of both the loop itself and of spawning new work is divided evenly along with the cost of the loop body.

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

5.3.2 Serial/parallel structure when spawning within a serial loop

Here is the DAG for a serial loop that spawns each iteration. In this case, the work is not well balanced, because each child does the work of only one iteration before incurring the scheduling

overhead inherent in entering a sync. For a short loop, or a loop in which the work in the body is much greater than the control and spawn overhead, there will be little measurable performance difference. However, for a loop of many cheap iterations, the overhead cost will overwhelm any advantage provided by parallelism.



5.3.3 cilk_for body

The body of a `cilk_for` loop defines a special region that limits the scope of `cilk_for` and `cilk_sync` statements within it. A `cilk_sync` statement within a `cilk_for` waits for completion only of the children that were spawned within the same loop iteration. It will not sync with any other iteration, nor will it sync with any other children of the surrounding function. In addition, there is an implicit `cilk_sync` at the end of every loop iteration (after block-scoped variable destructors are invoked). As a result, if a function has outstanding children when entering a `cilk_for` loop, it will have the same outstanding children when exiting the `cilk_for` loop. Any children that were spawned within the `cilk_for` loop are guaranteed to have synchronized before the loop terminates. Conversely, none of the children that existed before entering the loop will be synchronized during loop execution. This quality of a `cilk_for` loop can be used to your advantage (see Exception Handling).

If an exception is thrown from within a `cilk_for` loop body (and not handled within the same iteration), then some of the loop iterations may not run. Unlike a serial execution, it is not completely predictable which iterations will run and which will not. No iteration (other than the one throwing the exception) is aborted in the middle.

Windows OS: There are restrictions when using Microsoft structured exception handling (specifically, the `/EHa` compiler option and the `__try`, `__except`, `__finally` and `__leave` extensions to C/C++). See Windows* Structured Exception Handling.

5.3.4 cilk_for Type Requirements

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

Suppose the control variable is declared with type `variable_type` and the loop termination expression has type `termination_type`, as shown here:

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

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

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

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

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

```
variable_type::operator+=(difference_type);
```

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

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
```

5.3.5 cilk_for Restrictions

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

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

- 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. Therefore, 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();
  ```
- In C++, the control variable must be declared in the loop header, not outside the loop. The following form is supported for C, but not C++:
  ```
  int i; cilk_for (i = 0; i < 100; i++)
  ```
- A `break` or `return` statement will *not* work within the body of a `cilk_for` loop; the compiler will generate an error message. `break` and `return` in this context are reserved for future speculative parallelism support.
- A `goto` can only be used within the body of a `cilk_for` loop if the target is within the loop body. The compiler will generate an error message if there is a `goto` transfer into or out of a `cilk_for` loop body. Similarly, a `goto` cannot jump into the body of a `cilk_for` loop from outside the loop.
- A `cilk_for` loops may not "wrap around." For example, in C/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` loop.
- A `cilk_for` loop may not be an infinite loop such as:
  ```
  cilk_for (unsigned int i = 0; i != i; i += 0);
  ```

5.3.6 cilk_for Grain Size

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

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

5.3.6.1 Setting the Grain Size

Use the `cilk grainsize` pragma to specify the grain size for one `cilk_for` loop:

```
#pragma cilk grainsize = expression
```

For example, you can write:

```
#pragma cilk grainsize = 1
```

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

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

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

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

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

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

```
#pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
```

5.3.6.2 Loop Partitioning at Run Time

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

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

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

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

For example, consider a `cilk_for` loop of 16 iterations:

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

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

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

5.3.6.3 Selecting a Good Grain Size Value

The default grain size usually performs well. Use the following guidelines to select 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.

5.4 Preprocessor Macros

`__cilk`

This macro is defined automatically by the Intel compiler and is set to the Cilk version number. The value in this release is 200, indicating language version 2.0.

6 Cilk Execution Model

Cilk programming can seem a departure from traditional serial programming. Certain key concepts require explanation in order for you to write Cilk programs that perform and scale well.

The following concepts are key:

- **strands**: the structure of a Cilk program is best understood as a graph of strands connecting parallel control points.
- **work, span, and parallelism**: the expected performance of a Cilk program can be analyzed in terms of work, span, and parallelism.

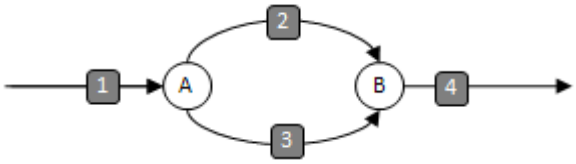
6.1 Strands

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

The word *strand* describes a serial section of the program. More precisely, the definition of a strand is "any sequence of instructions without any parallel control structures."

According to this definition, a serial program can be made up of many sequential strands as short as a single instruction each, a single strand consisting of the entire program, or any other partitioning. A *maximal strand* is a strand that is not part of a longer strand, that is, a strand whose start and end points are parallel control structures.

A Cilk program contains two kinds of parallel control structures - *spawn* and *sync*. (A parallel loop, i.e., `_Cilk_for`, is just a convenient notation for decomposing a problem into spawns and syncs.) The following illustrates 4 strands (1, 2, 3, 4), a spawn (A) and a sync (B). In this figure, only strands (2) and (3) may execute in parallel.

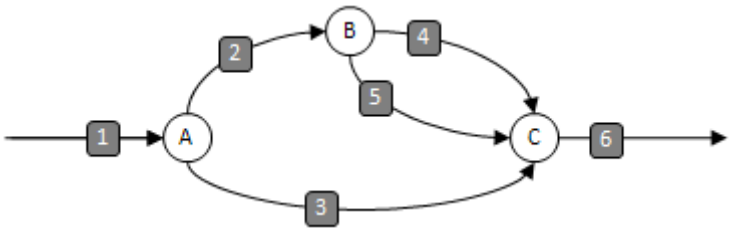


In this diagram, the strands are represented by lines and arcs (edges), while the parallel control structures are represented by the circular nodes (vertices). The strand diagram represents a Directed Acyclic Graph (DAG) that represents the serial/parallel structure of a Cilk program.

A Cilk program fragment that reflects the structure in the diagram is as follows:

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

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



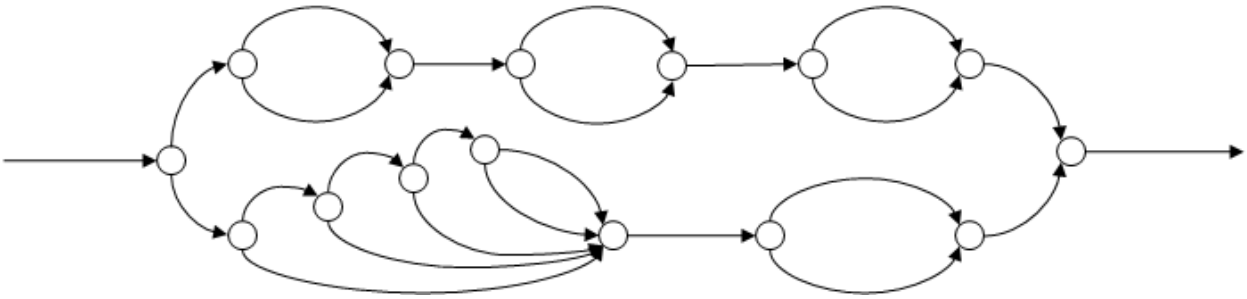
A DAG represents the serial/parallel structure of the execution of a Cilk program. With different input, the same Cilk program may have a different DAG. For example, a spawn may execute conditionally.

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

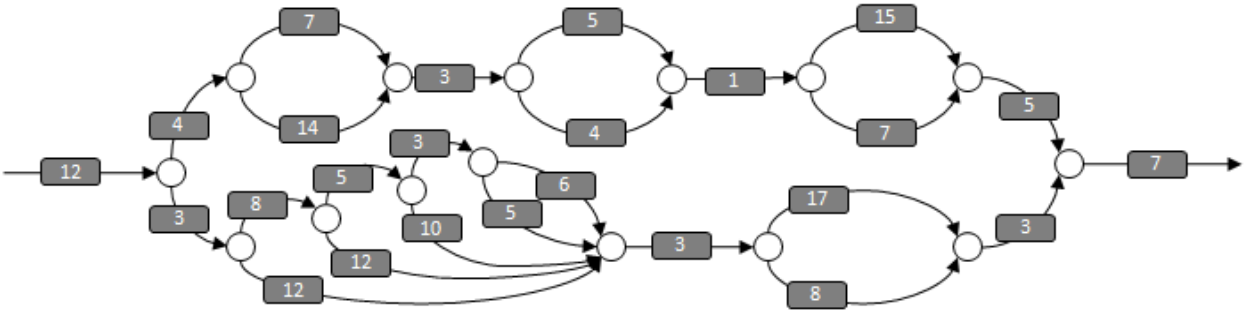
6.2 Work and Span

Once you have a method of describing the serial/parallel structure of a Cilk program, you can begin to analyze the program's performance and scalability.

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



This DAG represents the parallel structure of some Cilk program. To construct a Cilk program that has this DAG, it is useful to add labels to the strands to indicate the number of milliseconds it takes to execute each strand:



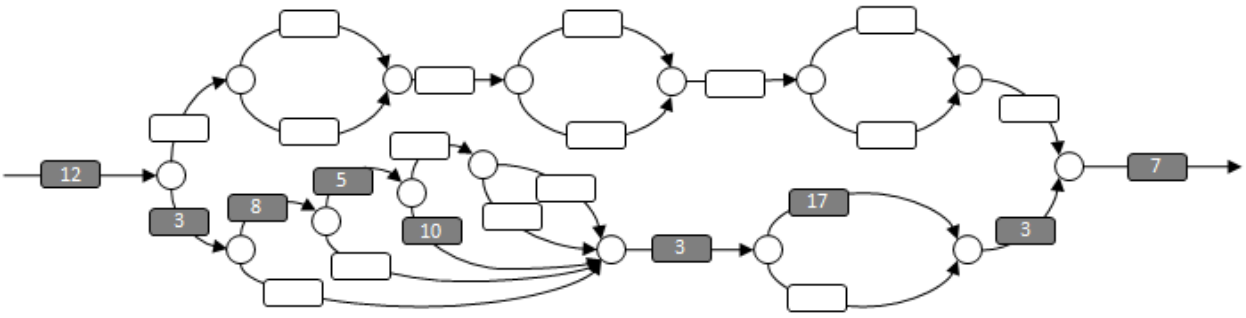
6.2.1.1 Work

The total amount of processor time required to complete the program is the sum of all the numbers. This value is defined as the ***work***.

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

6.2.1.2 Span

The ***span***, sometimes called the critical path length, is the most expensive path that extends from the beginning to the end of the program. In this DAG, the span is 68 milliseconds, as shown in the following diagram:



Under ideal circumstances (for instance, when there is no scheduling overhead) and when an unlimited number of processors are available, this program should run for 68 milliseconds.

You can use the work and span to predict how a Cilk program will speed up and scale on multiple processors.

When analyzing a Cilk program, you need to consider the running time of the program on various numbers of processors. The following notations are helpful:

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

T(1) is the Work

T(∞) is the Span

On 2 processors, the execution time can never be less than T(1) / 2. In general, the Work Law can be stated as:

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

Similarly, for P processors, the execution time is never less than the execution time on an infinite number of processors. Therefore, the Span Law can be stated as:

T(P) >= T(∞)

6.2.1.3 Speed up and Parallelism

If a program runs twice as fast on 2 processors, then the speedup is 2. Therefore, *speedup* on P processors can be stated as:

T(1) / T(P)

An interesting consequence of the speedup formula is that if T(1) increases faster than T(P), then speedup increases as the work increases. Some algorithms, for example, do extra work in order to take advantage of additional processors. Such algorithms are typically slower than a corresponding serial algorithm on one or two processors, but begin to show speedup on three or more processors. This situation is not common, but is worth noting.

The maximum possible speedup would occur using an infinite number of processors. *Parallelism* is defined as the hypothetical speedup on an infinite number of processors. Therefore, parallelism can be defined as:

$$T(1) / T(\infty)$$

Parallelism puts an upper bound on the speedup you can expect. If the code you are parallelizing has a parallelism of 2.7, for example, then you will achieve reasonable speedup on 2 or three processors, but no additional speedup on four or more processors. Thus, algorithms that are tuned to a small number of cores do not scale to larger machines. In general, if parallelism is less than 5-10 times the number of available processors, then you should expect less than linear speedup because the scheduler will not always be able to keep all of the processors busy.

6.3 Mapping Strands to Workers

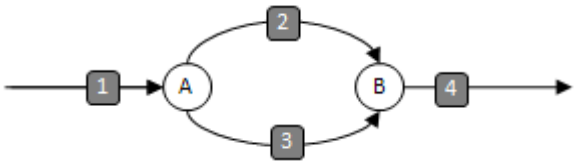
An earlier DAG illustrated the serial/parallel structure of a Cilk program. Recall that the DAG does not depend on the number of processors. The execution model describes how the runtime scheduler maps strands to workers.

When parallelism is introduced, multiple strands may execute in parallel. However, in a Cilk program, strands that *may* execute in parallel are not *required* to execute in parallel. The scheduler makes this decision dynamically.

Consider the following Cilk program fragment:

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

Here is the simple DAG for the code:



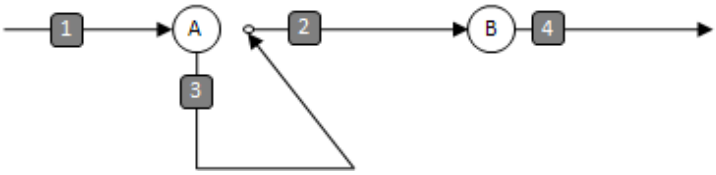
Recall that a worker is an operating system thread that executes a Cilk program. If there is more than one worker available, there are two ways that this program may execute:

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

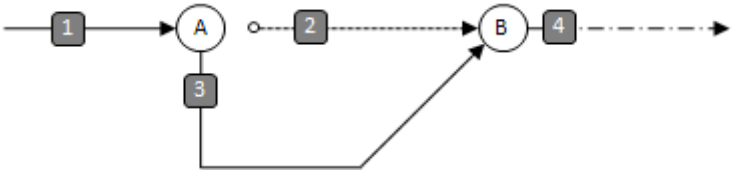
In order to guarantee serial semantics, the function that is spawned (the "child", or strand (3) in this example) is always executed on the same worker that is executing the strand that enters the spawn. Thus, in this case, strand (1) and strand (3) are guaranteed to run on the same worker.

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

To illustrate these two execution options, a new diagram is helpful. The diagram illustrates the execution on a single worker:



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



The details of the execution model have several implications that will be described in the section on the interaction between Cilk workers and system threads, and also the section on reducers. For now, the key ideas are:

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

6.4 Exception Handling

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

- 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.

Throwing an exception does not abort existing children or siblings of the strand in which the exception is thrown; these strands will run normally to completion.

If a `try` block contains a `cilk_spawn` and/or a `cilk_sync`, then there is an implicit `cilk_sync` just before entering the `try` block and an implicit `cilk_sync` at the end of the `try` block (after destructors are invoked). A sync is executed automatically before exiting a `try` block, function block, or `cilk_for` body via an exception. (Note that the scope of a sync within a `cilk_for` is limited to spawns within the same loop.) A function has no active children when it begins execution of a `catch` block. These implicit syncs exist to ensure that the same `catch` clauses run as would have run in a serial execution of the program.

Implicit syncs may limit the parallelism of your program. The sync before a try block, in particular, may prematurely sync a spawn that occurs before the try block as in the following example:

```
void func() {
    cilk_spawn f();
    try {  // oops! implicit sync prevents parallel execution of f()
        cilk_spawn g();
        h();
    }
    catch (...) {
        // Handle exceptions from g() or h(), but not f()
    }
}
```

If this is a problem for your program, there are a few ways to prevent the implicit sync:

- Move the body of your `try` block into a separate function. By moving the `cilk_spawn` lexically out of the `try` block, you eliminate the automatic generation of a `cilk_sync` at the end of the try block. The previous example could be rewritten as follows:

```
void gh()
{
    cilk_spawn g();
    h();
}

void func() {
    cilk_spawn f();
    try {  // no cilk_spawn in body, so no implicit sync
        gh();
    }
    catch (...) {
        // Handle exceptions from gh(), but not f()
    }
}
```

- Enclose the entire try/catch statement or just the body of the try block into a `cilk_for` loop that executes only once. The body of a `cilk_for` loop is an isolated context such that spawns and syncs do not interact with the surrounding function. If you do this a lot, a macro can come in handy. The above example could be rewritten as follows:

```
#define CILK_TRY cilk_for (int _temp = 0; _temp < 1; ++_temp) try

void func() {
    cilk_spawn f();
    CILK_TRY {  // try is within cilk_for, so does not sync f()
        cilk_spawn g();
        h();
    }
    catch (...) {
        // Handle exceptions from g() or h(), but not f()
    }
}
```

6.4.1 Windows* Structured Exception Handling

There are restrictions when using Microsoft Windows* structured exception handling (specifically, the `/EHa` compiler option and the `__try`, `__except`, `__finally` and `__leave` extensions to C/C++). Structured exception handling (SEH) will fail if an SEH exception is thrown after a steal occurs and before the corresponding `cilk_sync`. The runtime recognizes this condition and terminates the program with an appropriate error code.

7 Reducers

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

Cilk supplies ***reducers*** to address the problem of accessing nonlocal variables in parallel code. Conceptually, a reducer is a variable that can be safely used by multiple strands running in parallel. The runtime ensures that each worker has access to a private copy of the variable, eliminating the possibility of races without requiring locks. When the strands synchronize, the reducer copies are merged (or "reduced") into a single variable. The runtime 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 aprogram that uses reducers is the same as the serial version, and the result does not depend on the number of processors or how the workers are scheduled. Reducers can be used without significantly restructuring existing code.
- Reducers are implemented efficiently, incurring minimal overhead.
- Reducers can be used independently of the program's control structure, unlike constructs that are defined over specific control structures such as loops.

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

This chapter covers the following:

- Using one of the supplied reducers.
- Usage considerations, including appropriate operations, performance considerations and limitations of reducers.

At some point, you may want to write your own reducer. Appendix A: Writing Your Own Reducer provides more information.

7.1 Using Reducers — A Simple Example

This example illustrates use of reducers in accumulating a sum in parallel. Consider the following serial program, which repeatedly calls a `compute()` function and accumulates the answers into the `total` variable.

```
#include <iostream>

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

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

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

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

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

Converting this program to a Cilk program and changing the `for` to a `cilk_for` causes the loop to run in parallel, but creates a ***data race*** on the `total` variable. To resolve the race, you can make `total` a reducer; specifically, a reducer_opadd, defined for types that have an associative "+" operator. The changes are marked below.

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

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

```
}

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

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

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

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

The following changes in the serial code show how to use a reducer:

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

NOTE: 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.

7.2 How Reducers Work

An explanation of the mechanisms and semantics of reducers should help the more advanced programmer understand the rules governing the use of reducers as well as provide the background needed to write custom reducers.

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

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

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

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

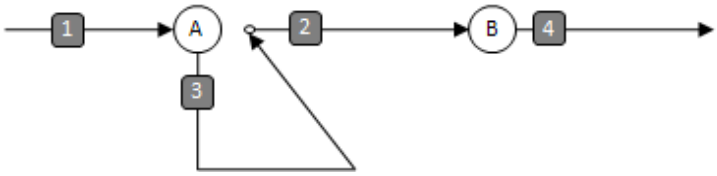
First, consider the two possible executions of a `cilk_spawn`, with and without a steal. The behavior of a reducer is very simple:

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

The following diagrams illustrate this behavior:

7.2.1.1 No steal

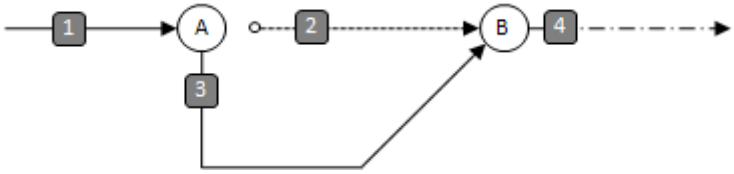
There is no steal after the `cilk_spawn` indicated by (A):



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

7.2.1.2 Steal

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



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

7.2.1.3 Example: Using reducer_opadd<>

This example consists of a simplified program that uses reducer_opadd<> to accumulate a sum of integers in parallel. For addition, the identity value is 0, and the reduction function adds the right value into the left value:

```
1  #include <cilk/cilk.h>
2  #include <cilk/reducer_opadd.h>
3
4    cilk::reducer_opadd<int> sum;
5
6    void addsum()
7    {
8        sum += 1;
9    }
10
11    int main()
12    {
13        sum += 1;
14        cilk_spawn addsum();
15        sum += 1;    // the value of sum here depends on whether a steal occurred
16        cilk_sync;
17        return sum.get_value();
18    }
```

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

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

Now, consider the case where a steal occurs. When `sum` is accessed at line 15, a new view with an identity value (0) is created. The value of sum after line 15 executes will be 1. The parent gets the new (identity) view and child gets the view that was active at the time of the spawn. This allows reducers to maintain deterministic results for reduce operations that are associative but not commutative. The child (addsum) operates on the leftmost instance, and so `sum` increases from 1 to 2 at line 8.

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

7.2.1.4 Lazy semantics

You can think of each strand as having a private view of the reducer. For performance purposes, these views are created lazily—that is, only when two conditions are met.

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

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

7.2.1.5 Safe operations

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

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

7.3 Safety and Performance Considerations

In general, reducers do not require locks to provide repeatable results across parallel runs. The results are the same as for a serial execution.

When developing your program, be aware of the following safety and performance considerations.

7.3.1 Safety

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

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

7.3.2 Determinism

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

7.3.3 Performance

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

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

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

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

7.4 Reducer Library

The provided reducer library contains the reducers shown in the following table.

A description of each reducer is described in the comments that appear in the corresponding header file.

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

REDUCER/HEADER FILE	IDENTITY/ UPDATE	DESCRIPTION
`reducer_list_append` `<cilk/reducer_list.h>`	empty list `push_back()`	Creates a list using an append operation. The final list has 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` `<cilk/reducer_list.h>`	empty list `push_front()`	Creates a list using a prepend operation.
`reducer_max` `<cilk/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` `<cilk/reducer_max.h>`	Arguments to constructor `cilk::max_of`	Finds the maximum value and the index of the element containing the maximum value over a set of values. The constructor argument has an initial maximum value and index.
`reducer_min` `<cilk/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` `<cilk/reducer_min.h>`	Arguments to constructor `cilk::min_of`	Finds the minimum value and the index of the element containing the minimum value over a set of values. The constructor argument has an initial minimum value and index.
`reducer_opadd` `<cilk/reducer_opadd.h>`	`0` `+=, =, -=, ++, --`	Performs a sum.
`reducer_opand` `<cilk/reducer_opand.h>`	`1 / true` &, &=, =	Perform logical or bitwise AND.
`reducer_opor` `<cilk/reducer_opor.h>`	`0 / false` |, |=, =	Perform logical or bitwise OR.
`reducer_opxor` `<cilk/reducer_opxor.h>`	`0 / false` `^, ^=, =`	Perform logical or bitwise XOR.
`reducer_ostream` `<cilk/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 is buffered by the reducer class until there is no more pending output to the left of the current position. This ensures that the output always appear in the same order as the output generated by the equivalent serial program.

`reducer_basic_string <cilk/reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Creates a string using append or `+=` operations. Internally, the string is maintained as a list of substrings in order to minimize copying and memory fragmentation. The substrings are assembled into a single output string when `get_value()` is called.
`reducer_string <cilk/reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Provides a shorthand for a `reducer_basic_string` of type `char`.
`reducer_wstring <cilk/reducer_string.h>`	Empty string, or arguments to constructor `+=, append`	Provides a shorthand for a `reducer_basic_string` of type `wchar_t`.

7.5 Using Reducers — Additional Examples

The following sections illustrate how to use a variety of reducers, including the String and List reducers.

7.5.1 String Reducer

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

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

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

The `cilk_for` loop uses a divide-and-conquer algorithm to break this into two halves, and then break each half into two halves, until it gets down to a "reasonable" size chunk of work. Therefore, the first worker might build the string "ABCDEF", the second might build "GHIJKLM", the third might build "NOPQRS", and the fourth might build "TUVWXYZ". The runtime system calls 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:

1. `"ABCDEF" concat ("GHIJKLM" concat ("NOPQRS" concat "TUVWXYZ"))`
2. `("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".

Cilk reducers provide serial semantics; these serial semantics are only guaranteed at the end of the parallel calculation, such as at the end of a `cilk_for` loop, after the runtime system has performed all the reduce operations. You should not call `get_value()` within the `cilk_for` loop; the value is unpredictable and meaningless since results from other loop iterations are being combined (reduced) with the results in the calling iteration.

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

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

int main()
{
    // ...

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

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

    return 0;
}
```

In this and other examples, each loop iteration only updates the reducer once; however, you can 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);
}
```

This is valid and produces the following string:

```
AaBb...Zz
```

7.5.2 List Reducer (With User-Defined Type)

`reducer_list_append` creates lists, using the STL list `append` method as the update operation. The identity is the empty list. The example here is almost identical to the previous string

example. The `reducer_list_append` declaration does, however, require a type, as shown in the following code.

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

int main()
{
    // ...

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

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

7.5.3 Reducers in Recursive Functions

The reducers are not limited to update operations within a `cilk_for` loop. Reducers can work with arbitrary control flow, such as recursively spawned functions. This example illustrates how a reducer can be used to create an in-order list of elements in a tree. The final list will always contain the elements in the same order as in a serial execution, regardless of the number of cores or how the computation is scheduled.

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

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

8 Operating System Specific Considerations

This chapter describes the following operating system specific considerations:

- How Cilk programs interact with operating system threads
- How Cilk programs interact with Microsoft Foundation Classes (MFC)

8.1 Using Other Tools with Cilk Programs

Because Cilk programs have a stack layout and calling conventions that are different from the standard C and C++ conventions, tools that understand the binary program executable (including memory checkers such as valgrind and code coverage tools) may not work with the parallel Cilk programs. Often, it is sufficient to run your program using only one worker (by setting the CILK_NWORKERS environment variable to 1). If that doesn't work, you can use such tools on the serialization of the Cilk program.

8.2 General Interaction with OS Threads

When working with OS threads, be aware of the following.

A worker thread is an OS thread.

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

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

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

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

Use caution when using native threading interfaces

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

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

- Do not use **Windows** thread local storage or **Linux*** Pthreads thread specific data, because the OS thread may change when work is stolen. Instead, use other programming techniques, such as the Cilk holder reducer described earlier.
- Do not use operating system locks or mutexes across `cilk_spawn`, `cilk_sync` or `cilk_for` statements, because only the locking thread can unlock the object.

8.3 Microsoft Foundation Classes and Cilk Programs

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

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

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

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

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

To add a computation thread to an MFC program:

1. Create a computation thread using operating-system facilities (`_beginthreadex` or `AfxBeginThread`). All the C++ code that is to be converted to Cilk should run in this thread. The computation thread leaves the main (UI) thread available to run the message pump for processing window messages and updating the UI.
2. Pass the handle (`HWND`) for the UI windows to the computation thread. When the computation thread needs to update the UI, it should send a message to the UI thread by calling `PostMessage`. `PostMessage` marshals and queues the message into the message queue associated with the thread that created the window handle. Do not use `SendMessage`. `SendMessage` is run on the currently executing thread, which is not the correct (UI) thread.

3. Test the C++ program to ensure that the logic and thread management are correct.
4. Add Cilk constructs to the logic in the computation thread.
5. Before terminating, the main (UI) thread should wait for the computation thread to complete, using `WaitForSingleObject()`.

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

Additional recommendations:

- When the main UI thread creates the computation thread, it should not wait for the thread to complete. The function that creates the computation thread should return to allow the message pump to run.
- Be sure that none of the data passed to the computation thread is allocated on the stack. If it is, it will quickly be invalidated as the computation-thread creation function returns, releasing the data.
- A data block passed to the computation thread should be freed by the computation thread when it is done, just before it sends a completion message to the UI.
- Use the `PostMessage` function instead of `CWnd::PostMessage`, as a primary reason for creating a computation thread is to avoid the MFC thread-local variables in Cilk code.
- A (short) computation using Cilk constructs can be called directly from the UI thread, as long as the computation behaves as a "black box" and does not attempt to communicate with the message pump or any other threads. This feature allows you to call non-interactive functions without regard to whether they use Cilk or not.

9 Cilk Runtime System API

Cilk programs require the runtime system and libraries, which are automatically linked into any program that uses Cilk. The runtime system provides a small number of user-accessible functions for controlling details of the program's behavior.

Include `cilk/cilk_api.h` to declare the Cilk runtime functions and classes. The entry points defined in this header file are described below. This header file does not define the interface to Cilk reducers. This interface is described in its own section in this book.

9.1 __cilkrts_set_param

```
int __cilkrts_set_param(const char* name, const char* value);
```

This function allows the programmer to control several parameters of the Cilk runtime system. The two string parameters form a name/value pair. The following `name` arguments are recognized:

`nworkers:` The `value` argument specifies the number of worker threads to use, as a decimal number or, if prefixed by `0x` or `0`, as a hexadecimal or octal number, respectively. If this function is not called, the worker count is obtained from the `CILK_NWORKERS` environment variable, if present. Otherwise, the worker count defaults to the number of processors available. This function is effective only before the first call to a function containing a `cilk_spawn` or `cilk_for`.

The return value is zero on success and a non-zero value on failure. Failures may be caused by a variety of conditions including an unrecognized argument, an illegal value, or calling at an incorrect time.

9.2 __cilkrts_get_nworkers

```
int __cilkrts_get_nworkers(void);
```

This function returns the number of worker threads assigned to handle Cilk tasks and freezes this number so that it can not be changed by `__cilkrts_set_param`. If called in serialized code, it will return 1.

The worker IDs are not necessarily in a contiguous range, so the ID for a given worker may be larger than the return value of `__cilkrts_get_nworkers`.

9.3 __cilkrts_get_worker_number

```
int __cilkrts_get_worker_number(void);
```

The function `__cilkrts_get_worker_number` returns a small integer indicating the Cilk worker on which the function is executing.

9.4 __cilkrts_get_total_workers

```
int __cilkrts_get_total_workers(void);
```

The Cilk runtime system may allocate more workers than are active at a given time. The function `__cilkrts_get_total_workers` returns the total number of worker ID slots, including those that are not actively in use. In other words, it returns one more than the maximum numeric ID that might be assigned to a worker. Typically, this number is a small increment over the number of actual workers assigned to run Cilk tasks. You can create an array of size `__cilkrts_get_total_workers()`, with each element indexed by a worker ID. As with `__cilkrts_get_nworkers`, this function freezes the worker count so that it can not be changed using `__cilkrts_set_param`. If called in serialized code, `__cilkrts_get_total_workers` will return 1.

10 Understanding Race Conditions

Races are a major cause of bugs in parallel programs. This chapter describes race conditions and provides programming techniques for avoiding or correcting them.

10.1 Data Races

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

A ***data race*** is a special case of a determinacy race. 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 operation. The program result depends on which strand "wins the race" and accesses the memory first.

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

For example, consider the following program:

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

void Strand1()
{
    a = 1;
}

int Strand2()
{
    return a;
}

void Strand3()
{
    a = 2;
}

int main()
{
    int result;
```

```
    cilk_spawn Strand1();
    result = cilk_spawn Strand2();
    cilk_spawn Strand3();
    cilk_sync;
    std::cout << "a = " << a << ", result = "
              << result << std:endl;
}
```

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

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

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

10.2 Benign Races

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

Here is a simple example:

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

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

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

10.3 Resolving Data Races

There are a number of ways to resolve a race condition:

- Fix a bug in your program
- Use local variables instead of global variables
- Restructure your code

- Change your algorithm
- Use reducers
- Use a Lock

10.3.1 Fix a bug in your program

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

10.3.2 Use local variables instead of global variables

Consider the following program:

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

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

int main()
{
    int idx;

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

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

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

```
int main()
{
//  int idx;                           // Remove global
    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            int idx = i*JMAX + j;      // Declare idx locally
```

```
            a[idx] = i+j;
        }
    }
...
}
```

10.3.2.1 Restructure your code

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

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

10.3.2.2 Change your algorithm

One of the best solutions, although not always easy or even possible, is to find an algorithm that partitions your problem such that the parallelism is restricted to calculations that cannot race.

10.3.2.3 Use reducers

Reducers are designed to be race-free objects that can be safely used in parallel. See the Reducers for more information.

10.3.2.4 Use locks

Locks can be used to resolve data race conditions. The drawbacks to this approach are described in Considerations for Using Locks. There are several kinds of locks, including:

- `tbb::mutex` objects from the Intel® Threading Building Blocks library
- system locks on Windows* or Linux* operating systems
- atomic instructions that are effectively short-lived locks that protect a read-modify-write sequence of instructions

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

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

int main()
```

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

Using a lock to resolve the race:

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

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

    return 0;
}
```

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

11 Considerations for Using Locks

You can implement various synchronization mechanisms in the hardware or operating system.

Cilk recognizes the locking mechanisms listed here.

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

The following lock terms and facts are useful:

- The following terms are interchangeable: "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.
- `tbb::mutex` is implemented using underlying OS mutex operations.

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

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

11.1 Locks Cause Determinacy Races

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

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

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

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

11.2 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.

Following is a simple example, with two strand fragments, where the aim is 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 is to acquire the locks in exactly the same order in both fragments; for example, switch the first two lines in Fragment B. You can release the locks in the opposite order, but doing so is not necessary to avoid deadlocks.

11.3 Lock Contention Reduces Parallelism

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

If you must use locks, consider observing these guidelines:

- Hold a synchronization object (lock) for as short a time as possible. Acquire the lock, update the data, and release the lock. Do not perform extraneous operations while holding the lock. If the application must hold a synchronization object for a long time, then reconsider whether it is a good candidate for parallelization. This guideline also helps to ensure that the acquiring strand always releases the lock.
- 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 ensures that the acquiring strand also releases the lock.
- Do not 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 ensuring 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.

11.4 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 as follows:

- 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:

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

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

void parent(int a, int b, int c)
{
    tbb::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 that 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" or “scoped lock” object won't help, because the guard object's destructor will not run until the catch block is exited.

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

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. In other words, 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 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.

12 Performance Considerations for Cilk Programs

Parallel programs have numerous additional performance considerations and opportunities for tuning and improvement.

In general, the Cilk runtime uses processor resources efficiently using a scheduling algorithm called work stealing. The work stealing algorithm is designed to minimize the number of times that work is moved from one processor to another.

Additionally, the algorithm ensures that space use is bounded linearly by the number of workers. In other words, a Cilk program running on N workers will use no more than N times the amount of memory that is used when running on one worker.

This chapter describes some of the more common issues seen in multithreaded applications such as those written in Cilk.

12.1 Granularity

Divide and conquer is an effective parallelization strategy, creating a good mix of large and small sub-problems. The work-stealing scheduler can allocate chunks of work efficiently to the cores, provided that there are not too many very large chunks or too many very small chunks. If the work is divided into just a few large chunks, there may not be enough parallelism to keep all the cores busy. If the chunks are too small, then scheduling overhead may overwhelm the advantages of parallelism.

Granularity can be an issue with parallel programs using `cilk_for` or `cilk_spawn`. If you are using `cilk_for`, you can control the granularity by setting the grain size of the loop. In addition, if you have nested loops, the nature of your computation will determine whether you achieve the best performance using `cilk_for` for inner or outer loops, or both. If you are using `cilk_spawn`, be careful not to spawn very small chunks of work. While the overhead of `cilk_spawn` is relatively small, performance will suffer if you spawn very small amounts of work.

12.2 Optimize the Serial Program First

The first step is to ensure that the C/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 resulted in 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.

12.3 Timing Programs and Program Segments

You should 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 -- preferably on a mix of different systems. 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 straightforward 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 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 may adjust clock rates to reduce power consumption and therefore reduce overall performance.

12.4 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**. If the grain size is too large, the program's logical parallelism decreases. If the grain size is too small, overhead associated with each spawn could compromise the parallelism benefits. The Intel compiler and runtime system use a default formula to calculate the grain size. The default works well under most circumstances. If your Cilk program uses `cilk_for`, experiment with different grain sizes to tune performance.
- **Lock contention**. Locks generally reduce program parallelism and therefore affect performance. Lock usage can be analyzed using performance and profiling tools.
- **Cache efficiency and memory bandwidth**. Covered later in this chapter.
- **False sharing**. Covered later in this chapter.
- **Atomic operations**. Atomic operations, provided by compiler intrinsics, lock cache lines. Therefore, these operations can impact performance the same way that lock contention does. Also, since an entire cache line is locked, there can be false sharing.

12.5 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 `matrix` and `matrix_multiply` examples cited in the Optimize the Serial Program First section.

A simple way to identify bandwidth problems is to run multiple copies of the serial program simultaneously, one for each core on your system. If the average running time of the serial programs is much larger than the time of running just one copy of the program, it is likely that the program is saturating system bandwidth. The cause could be memory bandwidth limits or, perhaps, disk or network I/O bandwidth limits.

These bandwidth performance effects are frequently system-specific. For example, when running the `matrix` example on a specific system with two cores (call it "S2C"), the "iterative parallel" version may be considerably slower than the "iterative sequential" version (4.431 seconds compared to 1.435 seconds). On other systems, however, the iterative parallel version may show 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 (including DRAM speed, number of memory channels, cache and page table architecture, number of CPUs on a single die). Be aware that such effects are possible and may cause unexpected and inconsistent performance results. This situation is inherent to parallel programs and is not unique to Cilk programs.

12.6 False Sharing

False sharing 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. Even though another core may not be using that data (reading or writing), it may 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 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. 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.

12.7 Memory Allocation Bottlenecks

The memory allocation system used by default on some Linux and Windows systems is known to be a bottleneck in parallel programs. When the program allocates or deallocates memory from the heap (e.g., using `malloc`, `free`, `new`, or `delete`) on these systems, the runtime library uses a mutex lock to prevent corruption of the heap data structures. Even on a single processor, this lock is quite expensive. However, when multiple strands try to allocate or deallocate memory simultaneously, the resulting contention on the lock can effectively kill much of the parallelism in the program. This means that the heap allocator does not scale to multiple processors.

One solution to this problem is to use a scalable memory allocator, such as the one provided by the Intel® Threading Building Blocks library (TBB). The TBB Scalable Memory Allocator is recommended for use with Cilk programs.

Appendix A How to Write a New Reducer

You can write a custom reducer if none of the supplied Cilk reducers satisfy your requirements.

Any of the supplied Cilk reducers can be used as models for developing new reducers, although some of these examples are relatively complex. The implementations are found in the `reducer_*.h` header files in the `include/cilk` directory of the installation.

Two additional examples are provided in the following sections.

Components of a Reducer

A reducer can be broken into 4 logical parts:

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

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

The Identity Value

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

- `0` is the identity value for addition:
```
x = 0 + x = x + 0
```

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

The Monoid

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

In the reducer library, a Monoid is defined by a type T along with following five functions:

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

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

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

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

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

Writing Reducers — A "Holder" Example

This example shows how to write a simple reducer; it has no update methods, and the `reduce` method does nothing. A "Holder" is analogous to the "Thread Local Storage", but without the pitfalls described in the General Interaction with OS Threads section.

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

Such a reducer has a practical use. Suppose there is a global temporary buffer used by each iteration of a `for` loop. This is safe in a serial program, but unsafe if the `for` loop is converted to a parallel `cilk_for` loop.

Consider the following program that reverses an array of `point` elements, using a global temporary variable `temp` while swapping values.

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

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

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

    void reset() { valid_ = false; }

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

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

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

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

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

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

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

    return 0;
}
```

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

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

The `point_holder` class is the reducer. It uses the `point` class as the view. The Monoid class contains a reduce function that does nothing, since it doesn't matter which version is retained. The rest of the methods of `point_holder` allow you to access and update the reducer data.

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

class point
{
    // Define the point class here, exactly as above
};

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

private:
    cilk::reducer<Monoid> imp_;

public:
    point_holder() : imp_() {}

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

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

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

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

To use the `point_holder` reducer in the sample program, replace the declaration of temp with a declaration using the reducer type.

Replace

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

with:

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

The rest of the original program remains unchanged.

To summarize how the `point_holder` reducer works:

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

Appendix B Reading List

The following documents provide additional information on Cilk features and concepts.

General Cilk information:

The ***Cilk Implementation Project site*** (http://supertech.csail.mit.edu/cilkImp.html) is a gateway to the MIT Cilk project. A ***project overview*** (http://supertech.csail.mit.edu/cilk/) with links to a set of three lecture notes provides extensive historical, practical, and theoretical background information. Cilk is based on concepts developed and implemented for the Cilk language at MIT.

The Implementation of the Cilk-5 Multithreaded Language (http://supertech.csail.mit.edu/papers/cilk5.pdf) by Matteo Frigo, Charles E. Leiserson, and Keith H. Randall

The Power of Well-Structured Parallelism (answering a FAQ about Cilk) (http://software.intel.com/en-us/articles/The-Power-of-Well-Structured-Parallelism-answering-a-FAQ-about-Cilk)

Serial Semantics:

4 Reasons Why Parallel Programs Should Have Serial Semantics (http://software.intel.com/en-us/articles/Four-Reasons-Why-Parallel-Programs-Should-HaveSerial-Semantics).

Examples:

Are Determinacy-Race Bugs Lurking in YOUR Multicore Application? (http://software.intel.com/en-us/articles/Are-Determinacy-Race-Bugs-Lurking-in-YOUR-Multicore-Application)

Global Variable Reconsidered (http://software.intel.com/en-us/articles/Global-Variable-Reconsidered)

These articles describe an older reducer syntax, but the concepts are still applicable.

Race Conditions:

What Are Race Conditions? Some Issues and Formalizations (http://portal.acm.org/citation.cfm?id=130616.130623), by Robert Netzer and Barton Miller.

This paper uses the term *general race* where we would say *determinacy race*. A *data race* is a special case of a determinacy race.

A solution to N-body interactions http://software.intel.com/en-us/articles/A-cute-technique-for-avoiding-certain-race-conditions, discusses a mechanism for breaking a computation into portions that can be implemented without races

Making Your Cache Go Further in These Troubled Times (http://software.intel.com/en-us/articles/Making-Your-Cache-Go-Further-in-These-Troubled-Times) discusses the issue of cache-friendly algorithms.

GLOSSARY OF TERMS

A

About the Glossary

The Glossary is an alphabetical list of important terms used in this programmer's guide and gives brief explanations and definitions.

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 and C++ programming languages that allow a programmer to express concurrency easily.

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.

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.

CPU

"Central Processing Unit"; 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

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.

G

global variable

A variable that is bound outside of all local scopes. See also ***nonlocal variable***.

H

hyperobject

A linguistic construct supported by the Cilk runtime that allows many strands to coordinate in updating a shared variable or data structure independently by providing each strand a different ***view*** of the hyperobject to different strands at the same time. The ***reducer*** is the only hyperobject currently provided.

I

instruction

A single operation executed by a ***processor***.

K

L

linear speedup

Speedup proportional to the ***processor*** count. See also ***perfect linear speedup***.

lock

A synchronization mechanism for providing ***atomic*** operation by limiting concurrent access to a resource. Important operations on locks include acquire (lock) and release (unlock). Many locks are implemented as a ***mutex***, whereby only one ***strand*** can hold the lock at any time.

lock contention

The situation wherein multiple strands vie for the same ***lock***.

M

multicore

A semiconductor chip containing more than one ***processor core***.

multiprocessor

A computer containing multiple general-purpose ***processors***.

mutex

A "mutually exclusive" ***lock*** that only one ***strand*** can acquire at a time, thereby ensuring that only one strand executes the ***critical section*** protected by the mutex at a time. Windows* OS supports several types of locks, including the `CRITICAL_SECTION`. Linux* OS supports Pthreads `pthread_mutex_t` objects.

N

nondeterminism

The property of a program when it behaves differently from run to run when executed on exactly the same inputs. Nondeterministic programs are usually hard to debug.

nonlocal variable

A program variable that is bound outside of the scope of the function, method, or class in which it is used. In Cilk programs, we also use this term to refer to variables with a scope outside a `cilk_for` loop.

P

parallel loop

A `for` loop all of whose iterations can be run independently in parallel. The ***cilk_for*** keyword designates a parallel loop.

parallelism

The ratio of ***work*** to ***span***, which is the largest ***speedup*** an application could possibly attain when run on an infinite number of processors.

perfect linear speedup

Speedup equal to the ***processor*** count. See also ***linear speedup***.

process

A self-contained ***concurrent agent*** that by default executes a serial chain of instructions. More than one ***thread*** may run within a process, but a process does not usually share memory with other processes. Scheduling of processes is typically managed by the operating system.

processor

A processor implements the logic to execute program instructions sequentially; we use the term "***core***" as a synonym. This document does not use the term "processor" to refer to multiple processing units on the same or multiple chips, although other documents may use the term that way.

R

race condition

A source of ***nondeterminism*** whereby the result of a concurrent computation depends on the timing or relative order of the execution of instructions in each individual ***strand***.

receiver

A variable to receive the result of the function call.

reducer

A ***hyperobject*** with a defined (usually associative) `reduce()` binary operator which the ***Cilk*** runtime system uses to combine the each ***view*** of each separate ***strand***.

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.

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.

serialization

The C/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/C++ program, will behave exactly as the original C/C++ program. The term "*serial elision*" is used in some of the literature. Also, see "***serial semantics***".

shared memory

Computer storage that is shared among several processors. A shared-memory ***multiprocessor*** is a computer in which each ***processor*** can directly address any memory location. Contrast with ***distributed memory***.

span

The theoretically fastest execution time for a parallel program when run on an infinite number of ***processors***, discounting overheads for communication and scheduling. Often denoted by T_∞ in the literature, and sometimes called ***critical-path length***.

spawn

To call a function without waiting for it to return, as in a normal call. The caller can continue to execute in parallel with the called function. See also ***cilk_spawn***.

speedup

How many times faster a program is when run in parallel than when run on one ***processor***. Speedup can be computed by dividing the running time T_P of the program on P processors by its running time T_1 on one processor.

strand

A ***concurrent agent*** consisting of a serial chain of instructions without any parallel control (such as a ***spawn***, ***sync***, return from a ***spawn***, etc.).

sync

To wait for a set of *spawned* functions to return before proceeding. The current function is dependent upon the spawned functions and cannot proceed in parallel with them. See also ***cilk_sync***.

T

thread

A *concurrent agent* consisting of a serial ***instruction*** chain. Threads in the same job share memory. Scheduling of threads is typically managed by the operating system.

throughput

A number of operations performed per unit time.

V

view

The state of a ***hyperobject*** as seen by a given ***strand***.

W

work

The running time of a program when run on one ***processor***, sometimes denoted by T_1.

work stealing

A scheduling strategy where processors post parallel work locally and, when a ***processor*** runs out of local work, it steals work from another processor. Work-stealing schedulers are notable for their efficiency, because they incur no communication or synchronization overhead when there is ample ***parallelism***.

The ***Cilk*** runtime system employs a work-stealing scheduler.

worker

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