Concurrent Data Structures in C++

CSInParallel Project

July 26, 2012

CONTENTS

1 Concurrent Data Structures in C++: Web crawler lab **1**
- 1.1 Your goals . . . 1
- 1.2 Source Code . . . 1
- 1.3 Packages needed . . . 1
- 1.4 Preparation . . . 2
- 1.5 To Do . . . 2

2 Introduction to STL containers **6**
- 2.1 Introduction . . . 6
- 2.2 The STL container vector . . . 7
- 2.3 The STL container queue . . . 8

CHAPTER

ONE

CONCURRENT DATA STRUCTURES IN C++: WEB CRAWLER LAB

In this lab you will complete some code provided to you that will 'crawl' the web from a beginning page to a given depth by following every linked page and scraping it for more links to follow. The links found on each page are kept in a data structure until they are processed.

1.1 Your goals

The goals for this lab are:

- complete and test a web crawler application, which fetches web pages then visits the links contained in those web pages, using STL containers;
- experiment with an example of threads programming, a type of multicore parallel programming;
- to complete a correct multi-threaded web crawler application that uses threaded building block (TBB) containers.

1.2 Source Code

Still need

The work on this lab requires a "tarball" named `cds.tar.gz`.
Instructors, please contact us for the complete code.

1.3 Packages needed

- C++ compiler
- Standard Template Library STL
- CURL library for web access
- Boost library, for threads and mutexes
- Intel's Threading Building Blocks (TBB)

- Make program

1.4 Preparation

Copy the tarball into your directory on a multicore linux machine. Then 'unxip' and 'untar' it like this:

```
% tar xfz cds.tar.gz
```

This will create a directory `cds` that contains the code. Change to that directory.

```
% cd cds
```

1.5 To Do

1. The directory `serial` contains several subdirectories, and is organized in a structure suitable for a software project that is capable of growing very large.

   Examine the code for this program. Observe the following:

   - The source files (`.cpp` implementation and driver modules) are contained in a subdirectory named `src`, and the header files (interface modules) are named with `.hcc` and are stored in a subdirectory named `include`.
   - Some of the state variables in classes within `serial` are STL containers, as described in a class handout.
   - Three classes are defined:
     - `spider` is the main class, with methods for crawling from page to page and for processing each page, and state variables for recording the work to be done (i.e., web addresses or *URLs* of pages to visit), the finished work (URLs already processes), and a `vector` of `page` objects.
     - `page` contains state variables for the URL of a particular web page (as a `string`) and a `vector` of URLs found in that web page (which are candidates for future processing). `page` also contains a method for scanning a web page and filling that `vector` with URLs that are contained in that page.
     - `raw_page` has helper methods for fetching pages from the web and for delivering the HTML code from a fetched web page.
   - The main program is in the file `spider_driver.cpp`. It obtains two values from the command line, namely the starting URL to crawl from, and the maximum number of URLs to visit (which must be *parsed* from a string to an integer). Then, the program performs the crawl and prints results.
   - The `Makefile` uses `make` variables to make it easier to change items such as compilation flags. Source files in the `src` subdirectory are compiled to produce object (`.o`) files in the `build` subdirectory, and those object files are linked to creat an executable named `bin/spider`.

   DO THIS: Write down other observations or comments about the `serial` code. Feel free to include opinions you may have about the code or its organization.

2. Comments in `spider.cpp` indicate that two parts of the code need to be filled in for the crawler program to work. Before actually fillin that code in, we will see if we can compile and run the code successfully.

   First, insert output statements in those two locations, to indicate whether those sections of the code are reached. One message might state "processing the page x" where x is the URL of the page being processed, and the other might state "crawling the page x".

   Then `cd` to the `serial` directory, and issue a `make` command in that directory `serial`. This should compile the code and create an executable `bin/spider`

3. Now fill in the missing code for the `spider::crawl()` method. *Notes:*

   - You will have to use appropriate methods for fetching a web page, processing that page, and adding that page to the finished `queue`, then adding 1 to the variable named `processed`.

     Note: The method for fetching requires a C-style string (null-terminated array of characters) for its argument, but the URL you are crawling is stored in a `string` object. Use the online documentation for `string` to look for a method of `string` that returns a C-style string with the same characters as that `string` object.

   - Do you get the expected output, given that `spider::process()` is still only printing a message?

4. Finally, complete the implementation of the `spider::process()` method, compile, and test. *Note:*

   - The method `spider::is_image()` currently always returns `false`. In a more sophisticated crawler, this method could examine file extensions in order to determine whether a URL is an image (no need to crawl) or not.

5. Change directory to the `cds/threads` directory within your lab1 directory.

   ```
   % cd ~/lab1/cds/threads
   ```

6. First, look at the sequential program `primes.cpp` in that directory. It computes a table of primes up to a maximum number, then counts the number of primes produced.

   `primes.cpp` does *not* use threads, but is an ordinary program with multiple loops.

   Then, compile and run the program, using the Makefile provided (`make primes`). Feel free to experiment. *Note:* Be prepared to wait...

7. Now, examine the multi-threaded program `threads/primes2.cpp`. This program uses the same algorithm for determining primes, but it divides the range 2..`MAX` up into `threadct` non-overlapping subranges or "chunks", and has a separate `boost` thread determine the primes within each chunk. Observe the following:

   - The loop for determining primes from 2 to `MAX`, which was located in `main()` in the program `primes.cpp`, has been relocated to a separate function `work()`. That function has two arguments `min` and `max`, and the loop has been modified to check only the subrange `min`..`max` instead of the entire range 2..`MAX`.

     Each thread will execute `work()` as if it were a "main program" for that thread.

   - `pool` is an array of pointers to `boost` threads.

   - Each `boost` thread is initialized with a unique subrange of 2..`MAX`. Threads other than the first and the last receive subranges of length `segsize`. The first and last subranges are treated specially, to insure that the complete computation covers exactly the range 2..`MAX`.

   - To construct a `boost` thread running `work()` with arguments `a` and `b`, the following constructor call would be used:

     ```
     thread(boost::bind(work, a, b))
     ```

   - The call above constructs a thread, but that thread doesn't begin executing until its `join()` method is called. Thus, there is a separate loop that calls `join()` for all the threads in the array `pool`, which starts up all the threads.

8. Now, compile and run the program `threads/primes2.cpp`, using the `Makefile` provided (`make primes2`). This program takes an optional positive integer argument, for the thread count (default is 1 thread).

9. You can time how long a program runs at the Linux or Macintosh command line by preceding the command with the word `time`. For example, the two commands

```
% time primes
% time primes2 4
```

will report the timing for running both the sequential `primes` program, and the multi-threaded `primes2` program with four threads.

Perform time tests of these two programs, for at least one run of `primes`, one run of `primes2` with one thread, and one run of `primes2` with 4 threads. Feel free to experiment further.

10. Examine the code for the program `parallel`. Observe the following:

    - The same four-directory structure is used as for the `serial` directory in the previous lab.
    - The header files `serial/include/page.hpp` and `parallel/include/page.hpp` are identical. You can use the following `diff` command to verify this:

      ```
      $ cd ~/SD/lab1
      $ diff serial/include/page.hpp parallel/include/page.hpp
      ```

      If the `diff` command finds any differences between its two file arguments, it will report those differences; if there are no differences, `diff` will print nothing.

    DO THIS: Use the `diff` command to compare `raw_page.hpp` and `spider.hpp` for these two versions `serial` and `parallel`. *Note:* The `diff` program will report differences by printing lines that appear differently in those files. Lines that appear only in the first file argument to `diff` will be prefixed by <, and lines that appear only in the second file will be prefixed by >.

    Here are differences between `serial/include/spider.hpp` and `parallel/include/spider.hpp`.

    - A different selection of `#include` directives appears in the two files. In particular,
      - `serial/include/spider.hpp` includes `<queue>`, for the STL queue container.
      - `parallel/include/spider.hpp` includes three other files instead of `<queue>`. One refers to a new file `atomic_counter.hpp` that is part of this program (in the directory `parallel/include`). The others provide two *TBB containers*, named `tbb::concurrent_queue` and `tbb::concurrent_vector`

        TBB stands for *Intel Threading Building Blocks*, which provides an alternative implementation of some common containers. TBB also provides various parallel algorithms, but we will not use those algorithm features in this lab.
    - The state variables `m_work`, `m_finished`, and `m_pages` use the TBB container types `tbb::concurrent_queue` or `tbb::concurrent_vector` instead of the STL containers `std::queue` and `std::vector`.
    - `parallel/include/spider.hpp` has two new state variables:
      - `m_processed`, which has type `atomic_counter` (defined in `atomic_counter.hpp`)
      - `m_threadCount` of type `size_t`, which is an integer type
    - In `parallel`, one of the constructors for `spider` has a second argument of type `size_t`.
    - There are two new methods, named `work()` and `work_once()`.

    Note: TBB containers are used instead of STL containers because TBB containers are *thread safe*. This means that multiple threads can safely interact with a TBB container at the same time without causing errors. STL containers are *not* thread-safe: with STL containers, it is possible for two threads to interact with a container in a way that produces incorrect results.

When the correct behavior of a program depends on timing, we say that program has a *race condition*. The parallel version of the program uses TBB containers in order to avoid race conditions. (Race conditions are discussed in other CSinParallel modules.)

Likewise, the state variable `m_processed` has the type `atomic_counter` instead of `int` or `long` because the `atomic_counter` type is thread-safe, enabling us to avoid a race condition that may arise if multiple threads interact with an integer variable at about the same time.

11. The files `serial/src/spider.cpp` and `parallel/src/spider.cpp` contain the main differences between these programs – the `parallel` version uses multiple threads. Running `diff` shows these differences:

    - In `parallel`, one of the constructors for `spider` has a second argument, to specify the number of threads to use.
    - The counter `processed` is a local variable in `spider::crawl` in `serial`. This local variable is replaced by a *state variable* `m_processed` in `parallel`.
    - The main work of `crawl` is moved into a method `work()` for the multithreaded version `parallel`, and that version creates `threadCount` threads to carry that work out. Note that `work()` has one argument, an integer index of a thread in the array `threads[]`.
    - The method `spider::process()` and the rest of the code in `spider.cpp` are identical (except for missing code). The comment within `process()` indicates that the same algorithm can be used for `parallel` as for `serial`. Why can the same algorithm be used in the multi-threaded version as in the sequential version?

12. Fill in the code indicated in two locations for the `parallel` version of `spider.cpp`, working from the code you wrote for the `serial` version, as indicated by comments in the `parallel` code. Then compile and run your program.

    Note: This version of the program requires three command-line arguments: the maximum number of URLs; the number of threads to use (this arg is new); and the starting URL to crawl.

    Run the program with multiple threads (say, 4 threads). What do you observe about the run?

    - You can examine the beginning of the output using the `more` program, e.g.,

      ```
      % bin/spider 100 4 www.stolaf.edu | more
      ```

      Each thread is programmed to print a message such as

      ```
      Thread 2 finished after processing 29 URLs
      ```

      when it completes.

    - You will probably find that a small number of threads processed all of the URLs, and that the other threads finished early without doing any work. How many threads processed URLs in your run? Can you think of a reason why the others finished early without processing any URLs? (*Hint:* Think about the work queue near the beginning of the program, just as the threads are starting their work.)

13. To spread the computational work out better among the threads, observe that the method `spider::crawl()` includes a call to a method `work_once()` that has been commented out.

    ```
    /* work_once(); */
    ```

    Remove those comments, in order to enable that call to `work_once();` . This will cause the program to process one web page before beginning multi-thread processing. If that first processed page includes several links, they will be added to the queue `m_work`, so that several threads can retrieve web pages to process when they first begin.

CHAPTER

TWO

INTRODUCTION TO STL CONTAINERS

Note: This reading refers to topics that we will not pursue in homework, but which may arise in general C++ coding. Such topics will be marked**(Extra)**.

2.1 Introduction

The C++ Standard Template Library*(STL)* is a software library of algorithms, data structures, and other features that can be used with any pre-defined C++ type and with user-defined types (that provide members such as a copy constructor and an assignment operator).

The STL uses a C++ feature called the *template* to substitute types throughout a class or function definition, which enables a programmer to avoid rewriting definitions that differ in the type being used.

For example, our IntArray class for homework provides a class structure around an array of int, with error checking, default initialization for array elements, whole-array assignment, etc. To define a class LongArray or FloatArray, we would only need to replace the type name int by a different type name long or float, etc. Templates make it possible to define a class Array<T> in which a type name T is used thrroughout the definition of that class, so that the notation Array<int> would specify a class that behaves just like IntArray, Array<float> would specify FloatArray, etc. Here, the angle brackets <...> are the syntax for invoking a template, and they enclose a type name to use throughout a definition.

A programmer can define her/his own templated classes. For example, here is a complete program with a definition of a templated structPair<T> that represents an group of two elements of the same type, together with an example use of that template.

```
   #include <iostream>
using namespace std;

template <class T>
struct Pair {
  T x;
  T y;

  Pair(T val1, T val2) { x = val1;  y = val2; }
};


int main() {
  Pair<float> p(3.14, -1.0);

  cout << p.x << ", " << p.y << endl;
}
```

The STL *containers* are templated classes defined in the STL that are designed to hold objects in some relationship. Examples:

- The array classes we have implemented in homework are examples of containers, although STL's array-like container vector<T> has some added useful features such as an ability to resize incrementally that our array classes did not include.
- Another STL container is called a queue<T>, and contains a linear list of elements that are added at one end and removed at another, comparable to a check-out line of customers at a grocery store. Unlike a vector, the elements of a queue do not necessarily appear consecutively in memory.
- **(Extra)** Another example is the map<K,T> container, which includes an operator[] that accepts "indices" of a type K (not necessarily an integer type) and associates values of type T with those "index" values.

This is not a complete list of STL containers, but illustrates some commonly used ones. See www.cplusplus.com.reference.stl for documentation of all STL containers

2.2 The STL container vector

STL's vector<T> is a templated class that behaves like an "elastic array," in that its size can be changed incrementally. For example, consider the following example program:

:: #include <iostream> using namespace std; #include <vector> int main() { vector<int> vec(4); vec[1] = 11; vec.at(2) = 22; vec.push_back(33); for (int i = 0; i < vec.size(); i++) cout << vec.at(i) << endl; }

The preprocessor directive

```
#include <vector>
```

tells the compiler what it needs to know to compile with vectors

The first statement is a variable definition that defines vec to be a vector of four integers. Each integer location is initialized at 0 by default.

The third statement uses the at() method of vector, and behaves exactly like operator[] except for a different style of error handling (at throws "exceptions" instead of crashing the program).

The call to vector's push_back() method appends a new element 33 to the end of vec

The resulting vector vec contains five int elements: 0, 11, 22, 0, and 33. Those values will be printed by the final loop.

A similar exercise could have been programmed with float, Dog, or another type.

(Extra) In the "exception" style of error handling used by methods such as at(), a runtime error (i.e., while the program is running, such as index out of bounds for a vector object) creates an object called an *exception* that includes information about that error. We say that the error condition *throws* the exception object. C++ provides an optional try/catch feature for capturing thrown exceptions and taking action on them; otherwise, throwing an exception causes a program to crash.

The vector method push_back() enables a programmer to extend the length of a vector object by one element, as shown above. That new element is added at the "back" or highest-indexed end of that vector.

Another vector method pop_back() enables a programmer to delete the last element of a vector, thus decreasing its length by 1. The methodpop_back() requires no argument and returns no value.

The vector index operator [ ] and the method at() both provide immediate access to any element in a vector.

The method size() returns the number of elements currently in a vector object.

The method back() returns the last element of a non-empty vector. Thus, vec.back() returns the same value as vec[vec.size()-1] orvec.at(vec.size()-1) .

The vector container provides methods for inserting or removing values at locations other than the back of a vector object, called insert()and erase(). However, these methods are not as efficient as push_back() and pop_back(). This is because vector elements are stored consecutively in memory, so inserting or removing an element at a position other than the back requires copying all element values from that position through the back value.

(Extra) Note that the methods *insert()* and *erase()* require iterator objects to specify position within a vector object. An *iterator* is an object that contains a pointer and has methods for certain pointer operations, such as a method next() for advancing to the next element in an array or vector.

See Also:

www.cplusplus.com/reference/stl/vector for a reference on vectors.

2.3 The STL container queue

STL's queue<T> is a templated class that is a *FIFO (first-in first-out)* container. This means that it is capable of holding an indeterminate number of elements of a particular type, organized in an ordered list, with each element added at one end of the list (the *back*) and removed at the other end (the *front*).

Whereas STL's vector templated class has many methods, a queue has only six specified methods (see www.cplusplus.com/reference/stl/queue):

- push(), which adds an element at the end of a queue,
- pop(), which removes an element from the beginning of a queue,
- front(), which returns the element at the front of a non-empty queue (next to be popped),
- back(), which returns the element at the back of a non-empty queue (most recent to be pushed),
- empty(), which returns Boolean True if there are no elements in a queue and False otherwise, and
- size(), which returns the number of elements currently in a queue.

Here is a code example of using a queue.

```
#include <iostream>
    using namespace std;
#include <queue>
int main() {

      queue<float> q;
      q.push(1.23);
      q.push(4.56);
      q.push(7.89);
      while (!q.empty()) {
        cout << q.front() << endl;
        q.pop();
      }
}
```

The output from this code should be the numbers 1.23 *then* 4.56 *then* 7.89, one per line.

An STL vector could be used in a situation where a *queue* would be appropriate (e.g., simulating a process comparable to a grocery-store checkout line), using the vector methods push_back(), front(), and erase() (to remove the front element). But a queue can be implemented more efficiently than a vector for this purpose, avoiding the copying of elements that are needed for vector's erase() method.

(Extra) The underlying data structure for a queue can be specified when that queue is created, using a second template parameter.

On the other hand, a queue provides no index or at() operator for accessing an element other than the front or back elements. The ability to access arbitrary element locations (e.g., via indices) is called *random access*, and if random access is needed, a vector may be more desirable than a queue.

As with vector, the templated container class queue can accept a user-defined type for its elements. Thus

```
queue<Dog> ,  queue<const Dog\*> ,
```

and other types may be used.