Programming for Performance (BSCS2011)

ASSIGNMENT 2: Bit Manipulation, Cache Effects

1 GENERAL INFORMATION and OVERVIEW

1.1 Learning Objectives

1. Write and use simple classes in C++.
2. Learn how to use new and delete for memory allocation/deallocation.
3. Use random number generation in C++.
4. Use `std::chrono::high_resolution_clock` for timing parts of code.
5. Be able to do bit manipulation in C++, using operators `<<`, `>>`, `&`, `|`, et c.
6. Be able to use more complex bit routines, such as `__builtin_clzl`, `__builtin_popcountl`, et c.
7. Become familiar with and make use of the `const` keyword.
8. Understand and implement some basic data compression and search algorithms.
9. Observe how cache and a program's working set size can affect runtime.

1.2 Deadline

Solutions must be uploaded to Moodle by Wednesday, 29 April, 23:59 (1 minute before midnight). You will lose 10% of the total marks for each 24 period that passes after the above deadline. Let Simon know ***early*** if you need an extension.

1.3 Rules (read carefully!):

1. Your submissions should be all your own work. No copying from peers or elsewhere. No cheating.
2. Apart from code you have developed yourself for the purpose of this course, use only the standard C++ libraries. No third-party libraries that you have downloaded from the Internet (or otherwise obtained) are allowed.
3. You should submit all your code in a single .zip archive, containing a single directory that contains all your code (subdirectories are allowed if you like). Your archive should contain a file called "README.Simon" that explains in detail how your code can be compiled and tested (i.e. run).
4. You may develop and test your code on whatever system you like, however, the code you submit must compile and run on the server **turso.cs.helsinki.fi**. You should make sure of this prior to submission. You should all have ssh access to this server from melkki.cs.helsinki.fi or melkinkari - contact Simon if you do not, for some reason.

1.3 Assignment Overview

Efficient manipulation of individual bits is a key aspect of space-efficient data structures that are used in systems in which memory is at a premium. For example, the intersection algorithms that are at the core of modern search engines do extensive bit manipulation. So do the data compressors used in modern database systems. So to do DNA read-mapping pipelines used in the study of genomes.

In this assignment you are required to:

1. Implement a BitArray class that provides methods to set and unset bits in an array of *n* bits (i.e. a bit array), with *n* specified at construction time. Apart from space taken by a reasonably small number of member variables, instances of the BitArray class should use at most *n* + 63 bits of memory and support setting and getting bits in constant time.
2. Support a function called *sum* on an instance of your BitArray class. The sum function should return the number of 1 bits (i.e. bits that are equal to 1) that appear in the bit array up to a specified position. The extra space used for supporting *sum* (in addition to that required for the instance of BitArray) should be at most *n* + 63 bits and the function should return an answer *sum* in constant time. *For this task, you may assume that the bits in the instance of BitArray will not change*. A suggested approach for sum is provided later in this document.
3. Implement a PackedIntegerArray class, which supports a sequence (array) of fixed-width integers and routines for getting and setting those integers. The class has two construction parameters *k* and *n*. The parameter *n* specifies the number of integers in the sequence. The second parameter, $k \leq 64$, specifies the *width* of each integer in the sequence: every integer stored in the packed integer array must be in the range $[0..2^k)$. Each integer can therefore be stored in *k* bits. Both *n* and *k* are specified at construction time and may not be changed afterwards. Apart from space taken by a reasonably small number of member variables, instances of your PackedIntegerArray class should use at most *kn* + 63 bits of memory and support getting and setting of integers in constant time.

In Assignment 3 we will combine the sum function with VByte codes for special effects.

1.4 Marks

Read the learning objectives at the start of this document and keep them in mind while working on your solutions. There are a total of 35 marks for this assignment:

- 10 marks for the BitArray task,
- 10 marks for implementing the rank function on your BitArray
- 10 marks for the PackedIntArray task, and
- 5+ marks for an “Exploratory Task” that you should argue for (as part of your report).

This last set of marks is obtained by you exploring different language features, doing things cleverly, or generally trying things out and discussing what you found. Marks will be awarded based on the difficulty and general interest of what you have tried, and the persuasiveness of your argument for getting the marks. Note that you can obtain more than 5 (and up to a maximum of 10) marks this way (and thus more than 35 marks for the assignment overall). Some suggestions for what you *could* try to obtain these marks are in Section 2, below. This is just a guide though, you should feel free to be creative.

2 PROGRAMMING and EXPERIMENTAL TASKS

Complete the coding tasks outlined in Section 1 above, some more details of which are provided below. The marks you obtain for each task will be given based on the code you produce (its quality, correctness, readability, comments) and the short report you write (10 pages total at the *absolute most*). In each task below, some specific things are asked for inclusion in your report, however your report can discuss any reflections or factoids related to the assignment you found interesting or want to mention.

Before You Start

The Chapter 8 of the guide *Tie Koodereksi* provides the very basics of bit manipulation:
https://tie.koodariksi.fi/cppe/8/
This will be a good starting point for some of you. It will likely be extremely useful for development/debugging to have your BitArray class implement a function that prints out it's bits. I have also put a brief discussion on bit setting and getting toward the end of this document.

This assignment requires you to generate random numbers (as part of testing and experimentation) and also to time your code. These two things are covered briefly in *Tie Koodereksi* too, so use that as a guide:
https://tie.koodariksi.fi/cppe/13/

Spend an intense day or so becoming familiar with basic bit routines as well as timing and random number generation functions before throwing yourself at the programming tasks of this assignment.

Task 1: Bit Arrays (10 out of 30 marks)

Implement the BitArray class, as specified above. At a minimum, your class should implement functions: get(i), which returns the value of the ith bit; and set(i,b), which sets the value of the ith bit to b.

What are the cache sizes (L1, L2, L3) for your development environment? What are the cache sizes on turso.cs.helsinki.fi server?

Do the following:

1. Write a program that accepts as a parameter a value *n* and uses this to construct an instance of your BitArray class.
2. Next, generate *m* random integers in the range [0,*n*] and save them in a std::vector. Iterate through the vector and set() those bits in your bit array to be 1. Record the total time taken for this setting process (not including the time to randomly generate the integers).
3. Iterate through the vector of positions again, and this time record the total time taken to call the get() function for each of the positions. Is there any asymmetry between the total times for set() and get()? Why do you think this is?

4. clear() your vector and generate a new set of *m* random positions. Iterate through them and record the total time for get() queries. Has the average time for a get() query changed? Thinking about the way cache memory works, try to explain what you observe.

Repeat the above process (of constructing a BitArray, then setting positions, then getting the same positions, then getting a different set of positions) for various different *m* values, something like, say *m* = 4096, 32768, 262144, 2097152, 16777216, 134217728. Possibly try different values of *n* too.

Task 2: Supporting the sum Function for Bit Arrays (10 out of 30 marks)

Recall from the assignment overview above that for a bit array B, the function sum(i) returns a count of the number of 1 bits in B[0..i]. The following is a possible and very simple (and slow (and unsafe!)) way to support sum:

```
uint64_t sum(const uint64_t i){
      uint64_t s = 0;
      for(uint64_t j = 0; j <= i){
            s += B.get(i);
      }
      return s;
}
```

Assuming the `get` function runs in constant time, the above method clearly takes O(i) time and uses no additional space.

Ideally, your solution to sum will take O(1) time (or, failing that, as close to constant time as you can get it), but you are allowed to use a small amount of extra space to achieve this. For the purposes of supporting sum, you may assume the contents of the bit array B will not change after your sum support structure is computed.

The suggested approach for supporting sum is to *store partial answers* to sum queries at evenly-space points in the bit array B. Let *t* be the distance (spacing) between the partial answers. The idea is to precompute and store the answers for sum(*t*), sum(2**t*), sum(3**t*), and so on. In total we will have (about) *n/t* precomputed answers (stored in an array).

Then, answering sum(*i*) simply a matter of first finding the nearest precomputed value before position *i*, say *x* = *j***t* and then looking at the bits in the bit array in between *x* and *i* to complete the answer.

This way, *sum* takes O(*t*) time. To get this down to constant time you will need to understand and make use of so-called builtin bit routines like: `__builtin_clzl`, `__builtin_ctzl` and `__builtin_popcountl`. Look these routines up online, understand what they do, and integrate them into your solution. Note that these function names are for gcc - people using, e.g., Visual Studio may need to find the names of equivalent routines (which should exist).

Run experiments to examine the performance achieved (i.e. total time taken for batches of sum queries) for various settings of *t*. What settings for *t* give better performance and why do you think this is?

Task 3: Packed Integer Arrays (10 out of 30 marks)
Write a program that takes two parameters:

1. n, the length of the packed integer array
2. k, the bit-width of each integer in the packed integer array

The program should first construct an instance of PackedIntegerArray with the specified *n* and *k*. It should then fill the packed integer array with random integers in the range $[0..2^k)$ and also save these values to a std::vector of uin64_t's. Your program should then verify that the contents of the packed integer vector are correct by checking that values obtained by get() are equal to the corresponding values in the std::vector of uint64_t.

Task 4: Exploration Task Suggestions (5 to 10 out of 30 marks)
Here are some suggestions you can choose from, with a rough guide for marks. Remember they are just suggestions: feel free to come up with your own ideas. The descriptions below are left deliberately vague (so that you still need to think a little bit).

1. Implement save() and load() routines for your classes that save and load instances of the classes to and from disk (1 mark).
2. Overload the [] operator for your BitArray class, so that you can get and set bits with it (rather than with just the get and set functions) (1 mark).
3. Look on the web and learn how you can do fast integer division (and modulus) by an integer *t* when you know that *t* is a power of 2. Speed up your *sum* support data structure by enforcing *t* to be a power of 2. (2-3 marks).
4. Use templates to implement your PackedIntegerArray or your sum support structure (5 marks).
5. Specialise the PackedIntegerArray for *k* = 8, 16, 32, 64. Compare the performance (i.e. time for get and set functions) to that of the class that works for general *k* (5 marks).
6. Implement rank to use (asymptotically) less than *n* + 63 extra bits (10 marks).

3 BRIEFLY ON BITS

This assignment involves thinking about bits. Bits are often hidden from us these days, but sometimes in order to write high-performance code we need to get our hands dirty with them.

The following line:

```
uint64_t x = 0;
```

gives us some bits like this in memory:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

I have put spaces between the bytes (8-bit blocks) for clarity.

In a picture like this, I usually think of the rightmost bit as being the least significant bit.

Anyway, this 64-bit integer can be thought of as a small bit array (one where n ≤ 64).

To set the 0th entry of the bit array to be 1, we could use the following code:

```
x = x | 1;
```

The above line computes the bitwise-OR of `x` and the value 1. After this, `x` would look like:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001

To set the 15th bit:

```
x = x | (1 << 15);
```

The `(1 << 15)` part of the line above shifts the value 1 by 15 bits to to the left, creating a value with the following bit pattern:

00000000 00000000 00000000 00000000 00000000 00000000 10000000 00000000

This is then bitwise-OR'd `x`, which would result in `x` looking like this:

00000000 00000000 00000000 00000000 00000000 00000000 10000000 00000001

To get the value of the 9th bit, we could use the following code:

```
uint64_t v = (x >> 9) & 1; //v contains 0 or 1
```

In our example, `v` would be equal to 0 (because the 9th bit of `x` is 0).

Of course, in your assignment, your BitArray class should handle larger n > 64. This will require you to manage an array of 64-bit integers, and work out how to find the appropriate element of that array to set (or get) the appropriate bit from that integer.

When implementing the PackedIntegerArray, you will essentially generalise the above ideas to work for chunks of *k* bits (rather than single bits, as is needed for the BitArray). To help get you thinking about this, if we wanted to extract the least significant block of 8 bits of x, we could do the following

```
uint64_t v = x & 255; //v is a value in the range [0..255)
```

This works because the bit pattern for 255 is

00000000 00000000 00000000 00000000 00000000 00000000 00000000 11111111

and `x`, as we know, currently contains

00000000 00000000 00000000 00000000 00000000 00000000 10000000 00000001

and the result of the bitwise-AND operation (specified by the `&` operator) means `v` contains:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001

If we instead wanted to know the value of the 2nd block of 8 bits, we could obtain it by first shifting `x`'s bits to the right by 8 bits, so that the 2nd block of 8 bits is now the 1st block of 8 bits, and then ANDing with 255 as before. Something like:

```
uint64_t v = (x >> 8) & 255; //v is in the range [0..255)
```

In general, to access the `i`th block of 8 bits:

```
uint64_t v = (x >> 8*i) & 255; //v is in the range [0..255)
```

To support a different block size to 8, then the `8` in the above line would need to change, and so would the `255`. Spend some time thinking about this. If you get *really* stuck, contact Simon - but do invest some time before deciding if you really need to.

4. TURSO RESOURCES

As specified in the Rules, the code you submit is required to compile and run on turso.cs.helsinki.fi. You are also welcome to use this environment for developing your code. Tools such as g++, grpof, perf, and valgrind are available there. You should be able to ssh into turso from melkki.cs.helsinki.fi and melkinkari.cs.helsinki.fi, and possibly from other servers (let Simon know early if you're having trouble).

A message from IT support:
$HOME has a very tight quota, but that is on purpose.

Students should use $WRKDIR for all their needs, this is the primary Lustre based FS which provides adequate performance. All others are very slow due to being NFS shared over gigabit ethernet, saturated over to hundred nodes.

$PROJ is for user applications etc, mostly for researchers to use for projects.

Documentation for using turso resources is available here:
https://wiki.helsinki.fi/display/it4sci/Scientific+Software+Use+Cases#ScientificSoftwareUseCases-0.0AVeryShortCoursetoABatchScheduling
The Ukko2 user guide deals with more complex circumstances:
https://wiki.helsinki.fi/display/it4sci/Ukko2+User+Guide